Automatic reproducibility and parallelism for biological image analysis workflows

A thesis submitted for the degree of
Doktor-Ingenieur (Dr.-Ing.)

Submitted to Faculty of Mathematics and Computer Science of the Friedrich Schiller University Jena, Germany

by Frank Taubert
born 04.09.1983 **in** Plauen

Reviewers

1. : Prof. Dr. Martin Bücker, Friedrich Schiller University Jena, Jena

2. : Prof. Dr. Philip Kollmannsberger, Julius-Maximilians-Universität, Würzburg

Day of defense: 25.11.2019

Abstract

Current microscopy techniques hugely profit from modern microscopes producing a massive amount of increasingly complex data which are analysed by sophisticated algorithms. As a result, previously undistinguishable phenomena can be observed. However, this development coincides with new challenges for the biologist executing these experiments. Data storage, data processing, parallelisation, automation, and reproducibility are important factors in mastering these new techniques as they incur additional effort previously of less impact for the biologists. Existing solutions address the mentioned factors separately. Image storage systems manage the storage of data, specialised tool solve individual processing problems, and workflow systems help with automation and ensure reproducibility. Finally, parallelisation is a topic that is slowly gaining traction in the field of the specialised tools. However, there exist gaps between these solutions that the biologist has to bridge by hand and which lower the overall efficiency. This work introduces a new software, whose design takes into account the mentioned aspects. It is a plug-in to the microscopy images storage system OMERO and is called OMERO Processing Extension (OPE). This approach eliminates nearly all of the overhead the biologist faces by integrating a system covering processing, reproducibility, and parallelisation into the data storage.

Zusammenfassung

Die Entwicklung neuer Techniken und Methoden in der computergestützten Mikroskopie haben die Grenze des Beobacht- und Messbaren immer weiter verschoben. Dabei basieren viele der heute verwendeten Methoden auf der komplexen Auswertung von großen Datenmengen. Daraus ergeben sich neue, anspruchsvolle Verarbeitungsschritte, die Wissenschaftler auf dem Gebiet der biologischen und klinischen Forschung auf dem Weg zum Endergebnis durchführen müssen. Diese zusätzlichen Schritte erschweren es dem Anwender sich auf seine Kernkompetenzen zu konzentrieren, da Aspekte, wie die Wahl eines angemessenen Verarbeitungswerkzeuges, die korrekte Verwendung von diesem, die Ablage der Ergebnisse sowie die Reproduzierbarkeit aller Schritte zu beachten ist. Lösungsansätze für einen Teil dieser Probleme sind in den letzten Jahren vermehrt vorgestellt worden. Es fehlt bis dato jedoch ein Ansatz, der alle Probleme in ihrer Gesamtheit adressiert. Für diesen Zweck wurde in dieser Arbeit OPE (OMERO Processing Extension) erstellt und im Folgenden untersucht. OPE ist eine Erweiterung für das OMERO Mikroskopiebildablagesystem. Es berücksichtigt von Grund auf alle angesprochenen Aspekte und befreit so den Nutzer von automatisierbarer Zusatzarbeit.

Danksagung

An dieser Stelle möchte ich mich bei all den Menschen bedanken, ohne die die Erstellung dieser Arbeit nicht möglich gewesen wäre. Dies betrifft insbesondere meinen Betreuer Prof. Bücker und alle Mitarbeiter des Lehrstuhls für Advanced Computing der Uni Jena. Insbesondere Daniel, Ralf und Thorsten, die sich immer wieder meine Ideen anhören mussten.

Dank geht auch an meine Freunde und Familie, auf deren Unterstützung und Rat ich mich stets verlassen konnte.

Nicht zuletzt möchte ich mich bei meiner Freundin Anne dafür bedanken, dass sie mich bei der Erstellung dieser Arbeit so ausdauernd unterstützt und ertragen hat.

Contents

1 Introduction

The recent technological progress of microscopy image acquisition and processing has greatly increased the amount of data generated in this field. These advancements produce more precise measurements, which make it possible to understand processes which could previously not be distinguished, deriving new knowledge.

However, these advancements come at a price. The acquired data has to be contextualised and processed to become information. Moreover, with the increased amount and complexity, the processing of this data becomes more challenging.

This development is amplified by advancements in data processing itself. With the growing understanding of the underlying processes, the algorithms to analyse the data get more sophisticated and specialised. As a result, it is challenging for a single researcher to understand all the algorithms available and used in the particular field.

Last but not least, the processing power of modern computers increases every year. As a part of this development a shift can be observed, away from single powerful processing units to highly specialised and massively parallel architectures (as can be found in graphics cards) as well as the use of compute clusters and cloud computing. These technologies also require new approaches and techniques to be used efficiently.

The problem with this development is that it gets harder and harder for a single individual to have a solid understanding of all the involved parts. This dilemma is especially true in disciplines like biology, where the preparation of the experiment itself requires a considerable amount of domain knowledge. To fully leverage all the current technologies a biologist would need to understand multiple fields: the underlying biological processes to design the experiment, the physical effect governing the data acquisition instruments as well as their technical limitations to optimally record the data, the assumptions and restrictions of the available analysis options to select an adequate algorithm, and finally the programming knowledge to use modern computers to run the analysis efficiently. And while some people master this whole spectrum, they should not be assumed to be the norm.

An additional concern, which has come into focus in the last years, is the desire to document experiments in such a way, that they can be easily be reproduced by other

scientists. To achieve this requirement, all involved steps have to be documented in a consistent and complete fashion. This reproducibility is not a novel concept; rather it is considered good scientific practice. Sadly, studies in recent years have shown that published research does not always live up to this standard [1, 2].

All these aspects offer an interesting challenge for computer scientists. One possible focus is the development of novel algorithms to run better, faster, and more efficient analyses. However, this endeavour is ultimately pointless if the users cannot correctly use these tools or are overwhelmed in their application. It should thus be of equal focus to provide solutions to lighten the workload and reduce the complexity of the tools that are being developed for collaboration partners.

Consequently, this work offers a solution that covers parts of the aforementioned problems, without increasing the complexity for the user. The focus is on scientists working with microscopes and biological samples, as the work is based on a collaboration with biologist through the German Research Foundation (DFG) funded Project INF of the CRC/TR 166 "High-end light microscopy elucidates membrane receptor function—ReceptorLight".

The following sections cover the available tools, which tackle the mentioned problems individually and investigates synergies and potential improvements. The following chapters analyse the operations needed by biologists (chapter 2) and the environment in which the software runs (chapter 3). The resulting requirements are used as a basis to conceive an adequate design (chapter 4) and explain the more intricate details implied by this design. These details are concerned with the distribution of the work (chapter 5) and the question how the system can be extended with additional functionality (chapter 6). Subsequently, the resulting software system is evaluated (chapter 7). Finally, a conclusion is drawn (chapter 8) and an outlook on future improvement possibilities is given (chapter 9).

1.1 Reproducibility

Before contemplating what new tools could be created, one should first try to see what old problems are still unsolved. One of the areas that could benefit from improvements is reproducibility, meaning the ability to repeat an experiment and come to the same conclusion [3, 4, 5]. Along with the related repeatability and replicability, these terms all centre on the same concept: a scientific result should be comprehensible from the experiment design all the way to the analysis of the results and the conclusion. This insight should be derivable using only the recorded

and published information, by either the experimenter at a later time or some other scientist. Only then, the results can be tested and verified. This principle is the foundation of the scientific discourse [6].

A way to achieve this reproducibility is to record provenance, meaning all the steps that have been performed on a given piece of data. Different disciplines have adopted different approaches to accomplish this idea. The creation of experiment protocols, lab books, and source control all try to preserve the essence, steps and tools used in an experiment. Peer review for publications is an additional step to ensure these qualities.

Still, the results are not always encouraging. Studies show that published results cannot always be verified by other scientists [1]. With a survey reporting that 80% of questioned researchers saw the current state of reproducibility as problematic [2].

The argument could be made that this flaw is system inherent, as the pressure to publish incentivises the publication of new data and penalises independent controls or negative results [7]. But this discussion is not the focus of this work. However, it should be noted that funding agencies try to address this problem by requiring more thorough recording of raw data. As a result, the questions of how data should be managed and reproducibility can be achieved are of growing importance in many fields. It is a desirable approach to achieve these goals through a computerised system if only to minimise the impact of new regulations on the scientists and the existing workflows.

There exists a still-evolving field of research discussing what has to be recorded to ensure reproducibility. Especially for experiments involving some step of data processing, it is required to record and share the data, the used software, and workflows [8], including software versions and computational environment.

Looking at the existing solution, it becomes apparent that the research is still in a phase, where separate tools try to automate and computerise different sub-aspects of a complete experiment workflow. In the next section, these sub-aspects are categorised and existing solutions are presented.

1.1.1 State of the Art

From discussions and interviews with collaborators the following rough steps of an experiment where generalised: experiment design, specimen preparation, performing the experiment, data acquisition, data preprocessing, data processing, analysis and result.

Experiment design: When designing an experiment the biologist first has to consider which hypothesis is tested and design an appropriate experiment. There exist a wide range of databases to support the experimenter in the selection of the correct materials used in the experiment. Examples are databases for fluorescence dyes, manufacturer specification sheets for proteins or vectors, chemical databases or databases for published gene sequences. Most of these databases also offer a unique identifier. These identifiers making unambiguous references to the used materials possible and improve reproducibility.

Specimen preparation and performing the experiment: In the next two steps, the experimenter has to prepare the biological sample and perform the experiment in accordance with the design. This step is historically recorded in a paper laboratory notebook (lab notebook/lab book/lab journal). This lab book is of great significance for the biologist. It contains detailed descriptions of each step of the process, taking precise note of what was used, how it was used, and how much of it was used. It contains the blueprint of the experiment and stays with the lab even after the scientist has left. Additionally, lab books have been used as evidence in legal cases.

The computerisation of the lab book is not a new concept and has apparent advantages: digital storage offers the possibility to access data from multiple devices, augment the experiment notes with images and references to the before mentioned databases, and assist with the recording of all needed information through input assistants. Moreover, the information can be easily duplicated and shared. With modern computers supporting touch inputs and digital pencils, writing and drawing are also possible [9]. The result is a user experience that can be as comfortable as the paper equivalent.

However, both legal and technical challenges make the widespread implementation of such systems complicated. From a legal standpoint, the paper versions are considered to be not trivially manipulable. Since, as a best practice, lab books have to be bound books any removal or change should be obvious. A digital version, on the other hand, has to implement mechanisms to convincingly and reliably track modifications, such as signatures and time stamps [10]. Further aspects, such as privacy, access control, or compliance with additional legal regulations apply depending on the nature of the experiment and national requirements.

Additional technical challenges arise, as the solution has to be usable in a lab environment, even during experiments. Depending on the lab, its safety regulation and the substances used, these requirements can be an additional obstacle. Last but

not least, electronic lab books may incur additional cost for licenses or hardware, which are potentially higher than for a paper book. Nevertheless, a respectable range of electronic lab books is available [11, 12].

Data acquisition: Data acquisition in biological experiments often involves, at least partially, capturing images. This step is either done with a camera or with pointwise scanning techniques such as Laser Scanning Microscopy, Atomic Force Microscopy, or Tip-Enhanced Raman Spectroscopy (TERS). Additionally, techniques such as Fluorescence Correlation Spectroscopy or the Patch Clamp Technique record non-image data[1]. Depending on the used instruments additional meta information may be automatically recorded. As an example, most modern microscopes offer the possibility to specify the dyes used in the experiment to assist in the correct configuration of the acquisition parameters. They will also write this information to the image file, as well as the state of all components they control. This meta-data is usually sufficient to reproduce the image capture step. However, for other measuring instruments the data can be as simplistic as a text file with voltage values, requiring the scientist to record from which experiment the data originated and what parameters were used in its acquisition.

The data captured in this step can be considered as the raw data of the experiment. Its storage is handled either by the user by hand or by adopting specialised storage software systems, such as OMERO[2] or BisQue. Using simple file system storage is easily implemented and well understood by the users. Use cases can also incorporate network shares or external hard drives. However, this approach implies that the experimenter has to record which file belongs to which experiment manually. Security concerns can also come into play when external (shared) devices are used.

On the other hand, dedicated storage systems like OMERO offer a structured way to archive the data. They also provide a means to view the data without the need to install additional tools. This solution simplifies sharing and collaborating. Some of the systems also include the possibility to run analysis scripts. Lastly, if the users are willing and able, most system support extension points for which the user can develop custom analysis and processing plug-ins and extend the functionality of the system as it suits their needs. Since this is a powerful and compelling way to tackle reproducibility, a more detailed introduction to OMERO, the system that was used

[1] See glossary for microscopy techniques.
[2] See software glossary for more information on mentioned software.

in the collaboration this work is based on, is given in subsection 1.1.2.

Data Preprocessing: The recorded images (as well as other forms of data) contain noise and artefacts created by the acquisition instrument. Techniques, such as deconvolution or de-noising, can further improve the quality and resolution. Hence, the image is usually run through a preprocessing step, before it is analysed. The tools used in this step consist of algorithms often already included in the image recording software such as ZEN or LAS, open source packages like ImageJ or Icy or commercial systems like Photoshop. This plethora of tools creates a number of problems. First of all, these steps incur a lot of manual labour for the experimenter. All the images have to be run through all the steps, often applying the same operation with the same parameters repeatedly without any automation. Secondly, different software packages may handle the provided metadata differently, possibly changing or removing it. Lastly, to record provenance all involved packages and plug-ins with all parameters and version numbers have to be recorded.

Data Processing: In the next step, the enhanced raw data has to be analysed. For simple images, this can mean counting of cells, measuring the size and intensity of a region of interest, or determining distances. As with preprocessing, this step uses many different tools, of which some are commonly used, while others are specifically written by the user. For non-image data, fitting a model or calculating specific characteristic values is a routine task. Like before, this requires accurate recording of versions and parameters to ensure reproducibility.

Data Analysis and Result: Finally, the scientist has to consider what implication the derived data has on the initial hypothesis and formulate a result. As part of the publication process, it is required to publish parts or all of the used data. Providing the provenance information needed by other scientists to reproduce the results is also necessary.

To correctly capture the provenance of the data and thus ensure reproducibility the experimenter has to create detailed notes, tracking all the steps that where performed. This effort is in addition to all the previously mentioned steps of the experiment execution. Ideally, as much of this tracking should be done automatically by the software used. Some tools already support this, such as the capturing software of many microscope manufacturers or workflow systems for processing data.

Workflow systems such as Taverna or KNIME try to alleviate some of the repetitive tasks involved in data (pre-)processing by formalising workflows. These workflows consist of inputs and outputs, as well as data transformation operations. The prevailing paradigm for the creation of these workflows is to use graphical programming, as it enables the user to create workflows without learning a programming language. Since workflows describe standard tasks, they ensure consistent results while simultaneously offering a formalised representation of the workflow (the workflow description) that can be used as an artefact to record provenance. These systems are also designed to be extended with a wide range of functionality, making them relevant for many problems. Last but not least, their automated execution reduces the workload for the user. This leads to them being widely used for automating tasks [13].

However, these systems do not come without problems. They are designed to be as flexible as possible. As a result, they are very generic and can be overwhelming when it comes to configuring a given workflow, making it difficult for a novice to create the desired workflow.

An option to reduce this complexity is to focus on one given domain. Using a more narrow scope allows a restriction to standard functions and the use of domain-specific terminology and iconography. Icy is one tool that implemented this approach for biological image processing. The presented work follows the same approach.

Another problem is the workflow description. While it accurately captures the workflow, it is another piece of data that has to be correctly attributed to an experiment result.

What is still missing is a tight integration of all these steps to span the complete experiment workflow. Image storage solutions offer a good starting point to explore the possibility of such an encompassing solution since the raw and final data is stored in such a centralised tool for long-term storage.

1.1.2 OMERO

The Open Microscopy Environment Remote Object (OMERO) [14] is a microscope image repository. It is being developed by Open Microscopy Environment (OME), an open source software initiative focused on biological microscopy data. OME also develops the popular Bio-Formats library for reading biological image formats, which is used in many tools, such as ImageJ or Icy.

Its primary function is to store images on a server and provide a convenient way

to display and share them. It also offers the capabilities to run analysis scripts, to search based on key-value pairs or tags, to help publishing the data, and to export images to different other formats.

Several tools are complementing these functions. These tools are implemented as comand line interfaces or plug-ins into the OMERO web interface. OMERO.insight and OMERO.dropbox for importing data, OMERO.Cli for command line interactions, OMERO.figure for creating publication images, as well as a number of plug-ins for different third-party packages which integrate into OMERO. The more recently added OMERO.parade and OMERO.mapr focus on searching and filtering data, emphasising the shift in development from merely storing data to providing data exploration and analysis capabilities.

The emerging focus on sharing can also be seen by their recent efforts concerning the Image Data Resource (IDR) [15]. The IDR uses OMERO to store and share data from published scientific studies in a publicly available image repository. IDR also includes the generation of reference Identifyers (IDs), such as Digital Object Indentifiers (DOIs). Such a repository is another way to ensure provenance of publication data. Furthermore, IDR extends OMERO to be used in conjunction with Jupyter notebooks. These notebooks are stored in a GitHub repository and are an additional way to record a processing step in a reproducible manner.

If the available OMERO features are not sufficient for a given use case, extension points for writing custom plug-ins. Such an extensions can contain a wide range of functionality, starting from simple custom analysis scripts and ranging to elaborate custom websites interfacing with OMERO.

An overview of the architecture is depicted in Figure 1.1. The server itself is based on Ice, a language-independent Remote Procedure Call (RPC) framework. Inside the server, various smaller services interact through these RPCs. Image data are stored on the file system. Image meta information and user data are stored in a relational database. As a design principle, image files cannot be modified.

While files are stored in a flat hierarchy on the file system, they logically belong to a group. Inside a group two more logical hierarchy levels exist, datasets and projects. Projects can only belong to a group, while datasets can either belong directly to the group or to a project in a group. The resulting hierarchy can hence be a maximum of three levels deep.

There also exist screens, which are a way to group multiple images that were created during screening experiments in which multiple images are acquired on a sample carrier contain multiple compartments. The sample carriers are used in

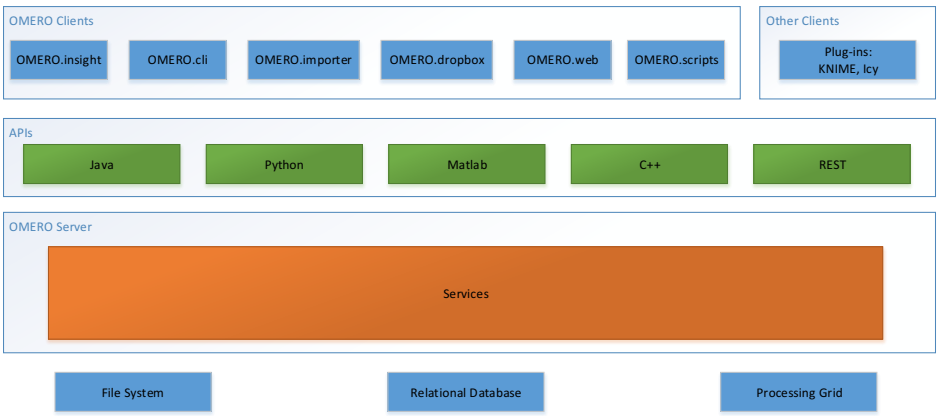


Figure 1.1: Omero architecture overview

high content screening applications, in which slight variations of the experiment are executed at the same time.

While such a comparatively shallow hierarchy might seem restrictive to a user habituated to the more flexible nature of file systems, this rigid structure fosters a shared data storage schema for all involved users.

Each of the OMERO objects (dataset, project, and image) can also be annotated. These annotations can be tags, tables, key-value pairs, and even files. Annotations are used to further describe an OMERO object as a way of improving reproducibility. Tags and key-value annotations are also used in the search functionality of OMERO.

The position of files and datasets in their respective root objects is established via links. This allows a single object to be present in multiple datasets without the need to be replicated in the file system.

The access to the server is handled via a range of Application Programming Interfaces (APIs), most of which are automatically generated by Ice. Those that are not, where hand created to provide a convenient way to access methods, but do not offer additional functionality.

On top of these APIs, a number of clients are realised. Most prominently the two rich clients OMERO.insight and OMERO.web, as well as OMERO.scripts.

OMERO.insight is a desktop client, which offers access to the full OMERO functionality, e.g. image import/export and display, searching, and running scripts. With recent releases (>5.3) the OMERO.insight development has switched to pure bug

fixing, with no more new features scheduled to release. This is due to the added workload of maintaining two separate clients.

OMERO.web is the second rich client to access OMERO. It is a web-based interface written in Python using Django web framework. Going forward, OMERO.web should be considered the primary way to access OMERO for most users.

Finally, OMERO.scripts is the scripting service of OMERO.web. The execution of a script is handled internally by the OMERO.grid, which allows the distribution of scripts onto multiple servers. The scripts used in this service are written in Python and specify which parameters are needed through a specialised function. At execution time, OMERO.scripts will auto-generate an appropriate Graphical User Interface (GUI) to pass the required parameters, ensure an active connection to the OMERO server, load required images, as well as take care of the generated output files. Besides that, scripts have little limitation and can be used flexibly to solve problems. However, the previously discussed reproducibility aspects are not automatically addressed and have to be handled by the creator of a script.

Overall, OMERO is a well-rounded storage solution for biological images. If paired with an adequate file system it can serve as a scalable long-term solution. Its recent releases focused handling bigger datasets and using OMERO as a repository for published data [15]. Paired with the fact that it is used in commercial applications it is very likely that long time support is ensured.

How OMERO can be extended to address reproducibility aspect is examined in chapter 4.

1.2 Biological High Throughput Computing

With advances in technology, microscopy generates more and more data. As an example, modern digital cameras offer a high-resolution and thus big images when capturing any kind of Wide-Field Microscopy image during an experiment. However, in comparison to some of the more specialised techniques, these datasets are still relatively small.

Exceptionally large datasets are created with techniques such as Photo-activated Localisation Microscopy (PALM) or Selective Plane Illumination Microscopy (SPIM), which record many camera images over the course of an experiment. Scanning techniques that produce many data points, often in the form of a spectrum, at once for each image pixel also produce large datasets. Examples for such techniques are TERS or spectral imaging.

The first concern when dealing with camera images is often to run preprocessing, such as de-noising or deconvolution (see 2.2.1), followed by aggregation methods, such as cross-correlation or image registration, or three-dimensional reconstruction from image stacks.

More specialised spectral images often require some form of (linear) unmixing. If the individual base spectra are not known, they have to be estimated or calculated using component extraction.

With datasets going into the terabytes, depending on the measurement method, the necessity of specialised and optimised algorithms is apparent. Alternatively, if the single experiment does not create that much data, techniques such as high throughput screening can run many experiments in a short amount of time, whose analysis poses its own challenges [16, 17].

This need for specialised tools and runtime optimisation is addressed by many of the software packages leading their respective fields: Huygens offers the possibility to run deconvolution on graphics cards, CellProfiler offers batch processing using a cluster, and most of the tools provide a Command Line Interface (CLI) for the integration into scripts.

However, these software packages address the issue only for their specific problem. A complete analysis workflow consists of multiple complex steps, each with its respective configurations and optimisation solutions. The need to run these steps in sequence, orchestrated by an automation tool, while ensuring that each tool can facilitate the maximum amount of optimisation, can lead to very complex and hardware specific workflows. Alternatively, the overall performance of the workflow degenerates as it provides only the minimum common ground for all tools to work. Finally, while optimisations such as using Graphics Processing Units (GPUs) are not unusual, distributing work over multiple calculation nodes is not commonly available.

Ideally, these tools should be integrated into a processing pipeline for easy access by the user. It will also be advantageous if the used parallelisation solution is generic enough to solve different kinds of problem.

In the following section, a sample of already realised solutions is presented. These solutions address either the generic usability or the convenient access. They also serve as an example to show the wide range of applications that deal with this problem.

State of the Art

As tools and their used algorithms mature, the frequency of fundamental changes slows down. This increased stability makes investing time to optimise performance more attractive. The results are incremental improvements in the existing, well established, image processing toolboxes. This optimisation can take the form of better data structures and algorithms, but also a better adoption of available hardware options.

In a comparable fashion, some advancements come from improvements in the underlying tools. For example, algorithms that have been developed and run in Matlab profit from improvements in Matlab itself. Similarly, all tools using Bio-Formats benefit from improvements in the library.

One popular hardware option used to improve performance is the use of multiple processor cores or GPUs [18, 19]. Listed in order of additional programming work needed, these improvements are achieved by using specialised compilers, appropriate algorithms, or libraries.

After the possibilities of the local machine are exhausted, turning to a distributed environment is the next logical step. OMERO can serve as a first example of how such a solution could look like. Here the distribution of scripts over compute-nodes is realised by the use of OMERO.grid. The approach constitutes running a distributed batch processing, meaning that a single job still utilises only a single node, but multiple jobs can be executed at the same time, each using a different node. The advantage of this parallelisation strategy is its low implementation complexity. Because the individual job is run on a single node, it does not require any kind of specialised code. When many jobs are present at the same time, the overall system uses its distributed calculation capabilities to finish faster than it would if everything would be running serial. However, if there are relatively few jobs to handle, parts of the system will lie idle, instead of speeding up running jobs. Still, this approach is quite convenient, as it offloads the computational work from the user's computer to a central server. Also, the shared resource is potentially more powerful. And while OMERO serves as a comfortable stand in, this approach is common practice for all use-cases employing a cluster resource management system to submit shell scripts for running analyses. Some solutions in the field of cloud computing also fall into this field, as they use rented computers such as Amazon Elastic Compute Cloud, Microsoft Azure, or Google Cloud Platform to perform their analysis on. While these solutions may raise some security and privacy issues, they are starting

to appear in academic publications [20, 21].

The last step in this series of optimisation approaches is to use multiple compute nodes to compute a single calculation, while simultaneously fully exploiting the hardware capabilities of each node, such as its GPUs. This is non-trivial, especially if it happens late in the development cycle of a tool, as the need for distribution can require a fundamentally different design of the tools. As such, true distributed computing is not found in any of the mentioned common tools. However, there exist attempts to achieve this goal. For example, solutions for distributed computing like Hadoop or SPARK are popular and easy to use.

Alternatively, utilising specialised frameworks exploiting inherent parallelisation opportunities are possible [22, 23]. This is also the approach that this work follows to achieve distributed parallelism.

To return to the previously explored topic of reproducibility, it is also desirable to completely integrate the distributed computation into a workflow system. This way a user can benefit from all the discussed optimisations without the need to learn the intricacies of each tool, while also using the workflow description as a reproducibility artefact.

Some systems exploring this idea are already available. However, they are most popular with screening applications, where parts of the experiment itself are also highly automised [24, 13, 25]. Additionally, these systems are custom built for a single application, with many specialised components interacting in particular ways. This design makes them inflexible and hard to adapt to new applications.

1.3 Automation Approaches without Programming

One of the most significant obstacles for automating a workflow is that describing the steps of the workflow is only possible with sufficient detail by writing code. Be it a small shell script to run some programs, a Python script to perform a simple analysis, or an R script to create some charts. Even though the involved tools and programming languages aim to be as easy to use, this step into the world of programming can be quite intimidating, as programming is no easy skill to learn [26]. However, the automation of mundane tasks is desirable as it allows the scientist to focus on the experiment. This section will present some of the solutions for this predicament.

State of the Art

Depending on the complexity of the actions that have to be automated, various solutions have been explored. As a simple solution, some tools provide a macro recorder. Such a recorder will document a sequence of user interactions and inputs while the user performs the task, which should be automated. This sequence is stored as a macro, which is a piece of automatically generated program code. Afterwards, the macro can be executed, automatically repeating the previously recorded user actions. Some examples of software systems which support such a mechanism are ImageJ, ZEN or Microsoft Excel. As an alternative, specialised standalone tools can automate mouse clicks on the operation system level. A disadvantage of this approach is the low flexibility. The user has to edit the created recording, as soon as a parameter has to be passed, a check has to be made, or a small modification is necessary. Such a modification leads back to requiring programming knowledge.

Another approach is to use visual programming. Here the program is not expressed as a text composed of a series of instructions. Instead, instructions are represented by a "box" and the interaction of these boxes forms the program. The graphical nature of this approach allows to communicate more semantics of an instruction that it is possible with words. There exists a number of graphical representations to visualise this concept [27]. Two popular methods are the representation using jigsaw pieces as instructions and boxes as instructions with arrows to represent data flow.

The representation as a jigsaw puzzle offers the possibility to express allowed data types as shapes. This way a user can easily distinguish which instructions and variables fit together. A downside is that more complex programs often degenerate into one big pile of boxes, which is hard to read. Nevertheless, programs like Google Blockly, Scratch, or Lego Mindstorms use this visualisation in an effort to teach programming and write small programs fast.

Representing a program as boxes connected with arrows is a popular paradigm used in workflow systems such as Taverna, Galaxy, or KNIME, but also in completely different disciplines such as instrument control (Labview) or animation (Unity, Blender). There are also workflow extensions that aim to provide workflow functionality to tools such as ImageJ [28] or Icy. Since the blocks are not as tightly packed as the jigsaw pieces and can be freely grouped, the appearance of a complete program is more structured. Datatypes are usually represented by colouring the blocks inputs and outputs of the same type in the same colour. Still, big programs can be hard to read and are plagued by arrows crisscrossing between the instructions.

1.4 Objective of this Work

The advantages of a system that incorporates all the aforementioned aspects are substantial. It allows the user to exploit the full capabilities of modern computers without the need to learn how to program and at the same time automatically ensure that the provenance information is correctly recorded.

There are already some promising approaches to achieve this goal, at least parts of it. The presented workflow systems solve much of the required reproducibility aspects and are easy to adopt as they commonly use graphical programming. However, they fall short when it comes to using the compute resources of modern machines. At best, they locally optimise single execution blocks. What they are lacking is the use of distributed computing resources.

Many of the individual processing tools exist in highly optimised variations and some can use distributed resources in a limited fashion. They can be integrated into workflows by hand, either through scripts or by extending existing workflow systems. But these solutions have problems preserving meta information, such as the used software version.

An optimisation which considers the whole workflow is currently only achievable by writing programs for languages that support these optimised distributed calculations, such as SPARK. However, this would require an extensive rewrite effort for existing tools and is challenging to offer without the need to require programming on the user side.

The approach furthest along [29] combines image and provenance storage (through OMERO) with graphical programming (through Galaxy) and blocks that individually use distributed resources (through map-reduce). However, it is still missing an overarching optimisation of the workflows.

This work introduces OPE, which takes the afore mentioned aspects into account from the beginning of its design. As a restriction, OPE will narrow its application focus to the field of biological image processing. This smaller focus results in a solution that is better tailored to the needs of the user. Two parts of this work have also been published. The first with a more narrow focus on the reproducibility aspects [30] and the second with a focus on parallism [31]

2 Operations

The target audience for OPE are biologists, performing analysis on microscopy images. This chapter will focus on the exploration of how and what processing operations are going to be used by most experimenters in this domain, since executing these operations is the primary purpose of OPE. More so, this understanding is needed to select representative calculations to evaluate OPEs performance. While this narrows down the potential general applicability, it allows the final system to better integrate with tools already adopted by biologists, more specific optimisation approaches, and a user experience which is easier to use.

2.1 Introduction to Biological Image Processing

Finding reliable data on the domain-specific operations used by biologists is not trivial, as there exist no general statistics on the topic. A possibility to grasp what operations are being used is to look at published software overview papers for biologists [17, 32, 33, 34, 35] or publications introducing a particular software including its design goals [19, 36]. The general groups that were identified are: preprocessing, segmentation [37], classification [38], and tracking [39].

Another approach is to look at the use of popular software packages and their operations. Here, it is rather difficult to get concrete statistics on which specific operations are used and how often. ImageJ, as well as Icy, record usage data. However, this data is only sparsely published. In the case of ImageJ, it can be employed to display where in the world the program is being executed. Icy's statistics page shows which versions are used, on what architectures, and how many plug-ins are listed in the plug-ins repository. This very conservative sharing of data is understandable, given the involved security and privacy concerns. Regrettably, it does not offer any insight into what operations the users of these programs are using.

Nonetheless, the Icy plug-in repository offers the possibility to rate a plug-in. These ratings can be examined, to give a rough indication of which plug-ins are

Plug-in Name	Short Description	# of Reviews	Category
Spot Detector	Detect, count, and calculate statistics of spots	17	Operation
Protocols	Allow using graphical workflows to automate operations	9	Automation
Channel Montage	View the individual channels of an image separately	7	User interface
Intensity Projection	Perform various projections through a stack or time series	7	Operation
Active Contours	Use manually drawn ROI to detect boundaries automatically	6	Operation
Manual Counting	Employ user interface extensions for manual counting	6	User interface
ROI Statistics	Calculate a large number of parameters for ROIs	6	Operation
Spot Tracking	Detect and track trajectories automatically	6	Operation
Image Browser	Browse images with thumbnails	5	User interface
Script Editor	Write scripts to extend the functionality	5	Automation

Table 2.1: Review statistics for Icy plug-ins [40], retrieved 26.02.2019

being used. The underlying idea is that plug-ins with a high number of reviews are commonly utilised. Naturally, these statistics only cover operations not provided by the core Icy functionality. All plug-ins with equal or more than five reviews are listed in Table 2.1.

The classes of plug-ins that can be found are evenly distributed over three general groups: user interface, operations, and automation. With the calculation plug-ins being: Spot Detection, Intensity projection, Region Of Interest (ROI) Statistics, Active Contours, and Spot Tracking.

This serves to underline the qualities biologists look for in the software they use:

- The possibility to add additional operations beyond the scope of the original system. Be it to add an improved version, test an own specialised algorithm or simplify the access to an existing one.
- The desire to have an efficient user interface, as shown by the various improve-

ment plug-ins in this group.

- The need to automate workflows, as shown by two different automation plug-ins in this group. Given the nature of the plug-ins (graphical programming and JavaScript), these workflows are relatively simple. The automation of biological image processing is also the focus of a number of publications [13, 28], as already discussed in section 1.2.

  One additional advantage of using automation is that the created code artefacts serve as a means to record precedence of data, enabling better reproducibility.

The data and assumed use patterns where confirmed in interviews with biologists. Next, the individual groups are introduced, along with some common operations found in them.

2.2 Preprocessing

Preprocessing focuses on improving the quality of an image, with the goal of preparing it for the actual processing that will follow. This includes reducing noise, adjusting colours, or deconvolution. Since many of the parameters in this step depend on the imaging device and not on the sample or processing that follows, even partial automation dramatically helps, as the same automated workflow can be used for all experiments using the same imaging device. The following operations are intended as a sample to gain some insight into what kind of operations belong to this category.

2.2.1 Deconvolution

Deconvolution is a technique for improving an image captured with a camera [41]. This makes it applicable not only for experiments capturing a single Wide-Field Microscopy image but also for techniques such as PALM or SPIM.

The underlining model represents an observed image $g(x, y)$ as the real image $f(x, y)$ that has been folded by a system inherent Point Spread Function (PSF) $h(x, y)$ and added noise. This PSF is the image resulting from observing a single point light-source through the imaging instrument. It is comparable to the impulse respone in signal processing systems and effectively blurs the image. If noise is negligible, than the observed image is generated by:

$$
\begin{aligned}
g(x,y) &= f(x,y) * h(x,y) \\
&= \sum_{(m,n)} f(n,m)h(x-n, y-m)
\end{aligned}
$$

Where:

$$x, y, m, n \in \mathbb{Z}$$

Deconvolution aims to remove the added blur by reversing the effects of the convolution. The PSF is either estimated or measured by imaging a suitable calibration sample. Since it is a property of the instrument and not of the sample, once captured, it can improve all the images acquired by the particular microscope. This general applicability is one of the reasons why performing deconvolution in real-time is an ongoing research topic [42].

Deconvolution works on single images or stacks. It can thus be run in parallel for multiple time points, z-positions, and channels.

2.2.2 Threshold

The next ubiquitous preprocessing technique converts a grayscale image into a binary image by applying a threshold. Here, the intensity value of each pixel of an image is set to either 0, if the value is less than a given threshold, or set to 1, if the value is greater or equal to the threshold. Alternatively, the minimum and maximum intensity of the used image format can be used as the resulting intensity value. This operation is essential for subsequently running tracking and segmentation algorithms. It can be formulated as:

$$
b(x,y) = \begin{cases} 0 & g(x,y) < t \\ 1 & g(x,y) \geq t \end{cases}
$$

Where:

$$x, y, t \in \mathbb{Z}$$

t Threshold

If the threshold value is known, this operation can be run for each pixel of an image in parallel. If not, it requires the analysis of the full XY-plane and potentially more dimensions, depending on the used algorithm. More advanced solutions for this problem will try to use different thresholds for different areas of the image based on

local information or try to identify intensity clusters [43, 44].

2.2.3 Histogram Linearisation

Histogram linearisation (or equalisation) is a method in which a single colour image with a given intensity distribution is transformed into another image of the same size and content, but with another colour intensity distribution.

In the original image, the spectrum of available intensity values is not entirely used, meaning that the minimum or maximum used intensity are unequal to the minimum or maximum value the image format supports. The transformation calculates a new intensity value for each value pixel based on its current intensity value. The structure of the image is not changed. This operation leads to a better contrast of the image, which makes it easier for a human to comprehend. A simple form to implement this operation is to stretch the original spectrum linearly to the full range:

$$g'(x,y) = G_{max}\frac{g(x,y) - g_{min}}{g_{max} - g_{min}}$$

Where:

$x, y \in \mathbb{Z}$	
$g_{min}, ..., g_{max}$	Intensity range used in the original image
$0, ..., G_{max}$	Intensity range supported by the image format

Similar to thresholding, this operation can be calculated independently for each pixel of all XY-planes, as it only requires the minimum and maximum intensity values of the plane. Alternatively, the minimum and maximum values of a stack, time point or the complete experiment could be used, depending on the necessity to keep the individual planes comparable for quantitative analysis.

2.2.4 Maximum Intensity Projection

Intensity projections are a group of operations in which the dimensionality of an image is reduced. For example, they can be used to convert a 3D image stack into a 2D image. The resulting image is useful for analyses such as finding areas of interest, as regions with high-intensity values can be identified without looking at each image of the stack, but rather the generated projected image containing all the maximum intensities. This standard implementation of a intensity projection is

called Maximum Intensity Projection (MIP).

Under the assumption that an XYZ-image stack should be projected along the Z-axis, the points of the resulting XY-projection-plane would be calculated as such:

$$g'(x,y) = f(g(x,y,0), ..., g(x,y,z-1))$$

Where:

$x, y, z \in \mathbb{Z}$

$f \quad \mathbb{R}^z \to \mathbb{R}$, max for MIP

In the case of the maximum intensity projection $f()$ returns the maximum of all passed arguments. Other popular function used for $f()$ are the mean or median.

Regarding a potential parallelisation, the Z-stack of every pixel can be run in parallel. However, this implementation can be a performance bottleneck, as most image types do not store these data points in a continuous memory area. Depending on the actual $f()$ used, this operation can also be expressed as a reduce-operation.

2.3 Segmentation and Classification

Segmentation is used to divide a given image into areas, which fulfil specific criteria. Depending on the criteria, this segmentation could be considered a preprocessing operation or the actual analysis. An example problem is the identification of contiguous areas of high intensity as a preparation for classification or to separate foreground from background. Classification, as a subsequent operation, attaches meaning to these areas.

An example where these operations are used is the detection of cells. This use-case requires the segmentation of the image into cells, followed by the classification of these cells into groups, such as healthy and unhealthy.

Both operations have been refined over a long time, as the underlying idea can be applied to many problems. Nowadays the employed algorithms are quite sophisticated, making it advisable to rely on existing tools, rather than reimplementing them. However, this incurs a strong dependence on in- and output-formats, making the efficient integration into a framework harder.

The degree to which these operations can be parallelised is immensely depending on the algorithm used. In the simplest case, each XY-plane is analysed individually and can hence be parallelised for all channels or time points. More complex

algorithms could use the different channels of an image, as these channels contain different structural information of the cell. A problem with the parallelisation of the classification is the output format of this operation. In contrast to the before mentioned operations, no image, but rather a table is created. Depending on the actual format, merging multiple sub-tables to form a complete result can be challenging.

2.4 Tracking

Tracking is used to record the position of the same object across multiple time points. As such it can require segmentation and classification to be run beforehand. Applications can be found when it comes to tracking how cells or particles move, but also in developmental biology to track which cells develop into what tissue.

As it has such a wide range of application, many tools have been developed to solve this problem [39]. When it comes to the parallelisation, these tools are difficult to integrate. As already hinted at when discussing classification, tracking does not produce images as an output, but potentially a tool specific format. Also, to track an object over the complete time span, it can be advantageous to have access to the complete time span. It is possible to run a partial tracking on parts of the time span and then merge the results. However, a longer observation time also means more information about the tracked object.

2.5 Parallel Structures

To implement parallelisation in OPE, the parallelisation possibilities of the before mentioned example operations have to be generalised. From a computer science perspective, almost all of the operations can be expressed either as a map, or as a reduce higher-order function.

The map-function can be used as a generalisation for operations where the complete image can be split into sub-images from which the sub-result can be computed without knowledge of any other sub-image. The composition of these sub-results then forms the end result. Operations falling into this category are: binarisation, counting cells, colour transformations, de-noising, deconvolution, edge detection, ROI-analysis, sharpening, and many more. However, there are some nuances between the operations in this list. First, regarding the outputs of the operations. Some of the operations produce images, while others produce a list. Furthermore, some of the

image producing operations preserved the size of the input image, while others modify it for the output. However, none of these operations removes an image dimension completely.

This reduction of the number of image dimensions is the domain of the reduce-function. This function can be used as a generalisation for operations where the complete image can be split into sub-images, that can be reduced pairwise (along a projection axis) into intermediate results, which are being further reduced until a single sub-result remains. The composition of these sub-results then forms the end-result. The reduce-function is commonly implemented to run in parallel using a tree-based algorithm [45]. Operations falling into this category are: (maximum) intensity projection, extraction of single time points (or channels, or slices), grayscale conversion, unmixing, as well as some super-resolution techniques.

2.6 Conclusion

It is hard to tell precisely what operations a biologist will use to analyse data. While some insight can be gained from studying popular tools and publication topics, reliable data are not readily available.

When considering the rough groups that can be found, it becomes apparent that many of the operations build upon each other, with a common core of preprocessing operations. This is encouraging, as preprocessing operations also have parallelisation potential, as most rely on local information.

With this in mind, the design of OPE focuses on how to parallelise preprocessing, where operations are relatively similar, require only local information and transform images into images. Most of these operations are expressible with either a map or reduce higher-order function. More complex operations seem to be too complex to reimplement them inside OPE. Therefore, the support of external tools in combination with map and reduce is also a goal.

3 Distributed Compute Environment

The execution of operations in a distributed compute environment is a major concern of OPE. As such an introduction into the available hardware and software is given in this chapter. Additionally a number of comparative benchmarks are executed to help with technology choices during the design process.

3.1 Cluster

This section will introduce the hardware that has been used while developing OPE. It should serve as a practical example of what options are available to a potential user.

3.1.1 Lofar Cluster

The Lofar Cluster is a distributed compute cluster of the chair for advanced computing of the University of Jena. When fully functional it consists of 17 identical compute nodes, 1 maintenance node and 1 head node.

The 17 compute nodes feature a dual socket mainboard equipped with two AMD Opteron 2378 quad Core chips, which are clocked at 2.4 GHz. All nodes have 16 GB of RAM. The are connected using Infiniband, with up to at $40\frac{\text{GB}}{\text{second}}$ and $1\frac{\text{GB}}{\text{second}}$ Ethernet.

Its purpose is to serve as a teaching environment for students. As such it offers a number of common big data and high throughput computing tools. For big data applications it supports Hadoop and SPARK. For High Performance Computing (HPC) applications, several implementations of the Message Passing Interface (MPI) standard are available (see subsection 3.2.1).

The Lofar Cluster is used as a test and debugging environment for the development of OPE. It also offers an OpenMPI version similar to the Ara Cluster (see subsection 3.1.2), it was used as a development environment, even if its performance

is considerably lower. This software basis, paired with full administrative privileges, made it the ideal development and test environment.

3.1.2 Ara Cluster

The Ara Cluster of the University of Jena is the new university-wide compute cluster. At the point of writing, it contains 272 compute nodes. This number is likely subject to change, as several extensions are planned and in various states of approval.

140 compute nodes feature a dual socket mainboard equipped with two Intel Xeon E5-2650 12 Core chips, which are clocked at 2.2 GHz. Two of these nodes have 1024 GB of RAM, all others have 128 GB. In addition, four of the nodes are equipped with NVIDIA Tesla P100 GPU cards.

The remaining 132 nodes were introduced later as part of an expansion. 128 use Intel Xeon 6140 18 Core CPUs at 2.3 GHz with 192 GB of RAM. Two addition nodes use Intel Xeon 6130 16 Core CPUs at 2.1 GHz and provide 1536 GB of RAM. As this part of the Ara Cluster was added during the development, it was not used for the benchmarks to preserve consistency over the full development period.

The nodes communicate via Intel Omni-Path, with up to $100\frac{\text{GB}}{\text{second}}$. The communication topology uses a 1:2 blocking fat-tree [46]. The nodes are arranged in groups of 32. Inside each group, all nodes can communicate with all other nodes of the group with the full bandwidth. However, only half of the group can communicate with other groups at full bandwidth at the same time.

As the Ara Cluster is used by a broad range of people, a wide range of software is available. Starting with standard packages, such as different implementations of MPI, to calculation packages like LAPACK, Matlab or Tensorflow. SLURM is used as the resource manager.

3.2 Software

To utilise both clusters, two software components are used. An implementation of Message Passing Interface (MPI) is used to write a program that runs in a distributed fashion. The workload manager is needed to queue that programs execution on the distributed resources.

3.2.1 MPI

The Message Passing Interface (MPI) is a low-level programming interface for developing distributed applications. It is standardised by the MPI Forum and has been implemented in prominent packages such as Intel Parallel Studio, OpenMPI or MPICH.

As the name implies, MPI's programming model focuses on the concept of messages. At its core, MPI will send data to or receive data from other nodes by sending the data as such a message. The standard also contains additional methods dealing with particular constellations more efficiently, such as broadcasting the same message to all nodes or scattering elements of an array. The messages only deal with blocks of binary data. Deriving meaning from the send data is up to the programmer.

By flowing these principles, MPI provides a low-level abstraction to network communication. Because of the focus on the bare minimum, it is considered to have low overhead and to be very fast.

To execute an MPI application, it is first distributed over multiple worker nodes. This requires a local copy of the program on each involved node, which is realised in the used clusters by means of shared network storage, which is synchronised over all nodes. Then, the node executing the initial start command remotes on each involved node and starts the application. When initialising MPI individually, all these applications on the different nodes will establish the communication between each other. They will then proceed to execute the programming and interact with each other until all instances are finished.

3.2.2 Workload Manager

As discussed above, running a distributed application can be as simple as starting the same application on all involved nodes. While this is a viable option if few people use the resources, it becomes inefficient as more and more people start their applications on nodes without checking what is already running.

To fix this problem workload manager, such as Yet Another Resource Negotiator (YARN) or Simple Linux Utility for Resource Management (SLURM), work as an arbiter to manage which application is executed when and where. In such an environment the user has to submit a job, including a list of constraints, to the workload manager. The manager then determines which of the appropriate nodes are free and execute the job using them. If this is not possible, the job is delayed

until enough other jobs have finished.

These managers also include many convenience features such as prioritising jobs, ensuring fair use, and limiting compute time. Because of these capabilities, larger clusters routinely require their user to use such a manager.

One downside of using these managers is that the user potentially has to wait until the job is started, which can have implications on how the use of a cluster is integrated into other systems.

3.3 Benchmarks

To understand the capabilities of the used hardware and to acquire a performance baseline, it is paramount to measure how fast the cluster can perform certain operations with a given technology. The following section will thus outline benchmarks that reflect use cases, which OPE will likely face. The results are used to decide what technologies are used for the implementation of OPE. More specifically: it was an early consideration to use Java for parts of the implementation, as it supports describing algorithms using a high level of abstraction. This abstraction makes Java convenient when writing complex systems. However, as an alternative, C++ is the native language for writing MPI programs and has a reputation for being efficient. Therefore, before committing to a language, it is crucial to know what performance impact this choice will have.

There are some publications comparing Java [47] and Python [48] MPI implementations to pure C++ MPI implementations. The general results are encouraging, as all three languages can achieve comparable performance. The library selection for Python is small. The mpi4py Python library is broadly used, actively maintained, and has good documentation. Most other libraries are either inactive or explicitly abandoned.

The selection for Java is much broader. However, overall documentation is not extensive and many projects either never left the prototype stage or have since become inactive. There also exist two types of implementations. Some projects settle for wrapping native C MPI libraries using the Java interoperability mechanism, while others implement the MPI standard using pure Java. Plain wrappers are relatively easy to write, promise little overhead, and are included in some of the available MPI packages. Pure Java implementations offer a better support for Java classes. However, the evaluated options seemed to be inactive and had inadequate documentation. Because of this state, the wrapper shipped with OpenMPI was

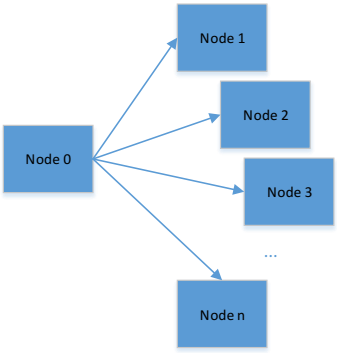


Figure 3.1: Centralised distribution benchmark

selected for the following tests, as it is decently documented and part of a widely used implementation.

All benchmarks were run on the Ara Cluster (see subsection 3.1.2). The execution was scheduled to nodes number 10 and following to ensure comparable results.

3.3.1 Centralised Distribution Benchmark

OPEs primary mechanism to access data is to load images from a file into the system, eliminating any direct dependencies on OMERO. During this process of loading the date, each image is exclusively handled by a single node. The notion of loading each input image with all nodes to reduce transfer time was rejected for now, as the reduction in data transfer would be only beneficial for the first calculation.

As a result, the complete image is stored in the memory of a single worker node. This means that the following computations on other nodes will have to request parts of the data from this node.
To measure a baseline for this kind of transfer a micro benchmark outlined by the following steps (compare Figure 3.1) are used:

1) Create n MPI worker nodes.

2) Node 0 will reserve x Bytes of data.

3) Node 0 will send these x Bytes sequentially to the other nodes and measure the time needed.

4) To reduce the variations in the time measurement, this is repeated multiple time. The measured throughput is the average of all these measurements.

The factor that is varied is the data size x, as it will show how well the system handles different data sizes. Different numbers of repetitions of the measurement were also tested to see if Java Just in Time (JIT) compilation has an impact on the performance. The benchmark was performed using eight nodes. Comparing different numbers of nodes (between 4 and 32) in a small preliminary test did not change the results. This is to be expected, as only the sender and one receiver node are using the network at any given time.

Results

While experimenting with different implementations, it became apparent, that the used data type has a significant impact on the achieved throughput. The results of this comparison are presented in Figure 3.2. The X-axis shows the size of the sent data in Byte, while the Y-axis shows the throughput in $\frac{\text{MB}}{\text{second}}$. The different colours represent the different implementations. The line type is used to separate the beginning of the measurement, where the time needed is constant and the performance thus limited by overhead, from the later part where it is limited by the actual library performance. Each measurement in Figure 3.2 was repeated at least 1000 times.

The first observation that can be made is that both the Java and Python bindings can achieve a performance close to the native C++ library that they wrap. At roughly $113000\frac{\text{MB}}{\text{second}}$, or $90.5\frac{\text{GB}}{\text{second}}$, the peak performance is close to the $100\frac{\text{GB}}{\text{second}}$, which is the upper bound set by the hardware used in the Ara Cluster (see subsection 3.1.2).

New Input Output (NIO) direct buffers have to be used to achieve this performance with Java. These are highly optimised for native I/O operations and "may reside outside of the normal garbage-collected heap" [49]. This means that the C++ commands underlying the Java wrapper can directly access this memory.

The other interface provided by the Java Bindings uses the `Object` class as the parameter type. Even in the simplest case of using this interface with a byte array, the throughput is comparable only for small data sizes. The throughput drops significantly for larger chunks of data.

Another factor that can influence Java performance is the JIT compiler. Its impact was measured by executing the measurements with different repetition counts. The

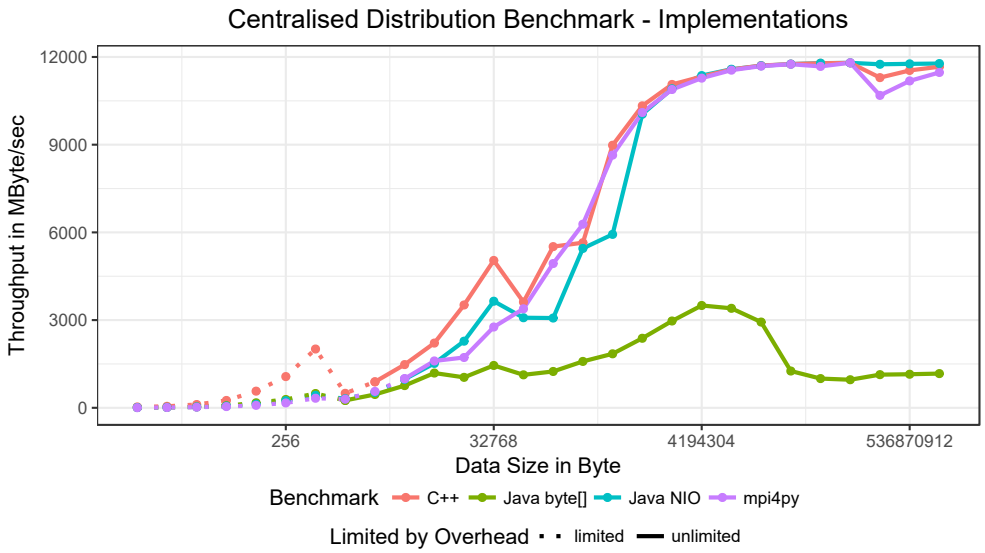


Figure 3.2: Different implementations of the centralised distribution benchmark. 8 nodes and 1000 repetitions

results are shown in Figure 3.3. The effect is minimal when using NIO buffers. In comparison, when using garbage collected byte arrays the increase in performance is considerable.

Figure 3.3 also shows a massive drop in performance for data sizes greater than 4 MB when using byte[], regardless of the repetition count. This is most likely rooted in the wrapper, which has to allocate a direct buffer for each call. As stated in the Java documentation [49] these buffers are initialised with 0, which would explain that the remaining throughput is nearly constant.

The Java bindings also have a few quirks one should be aware of. Their documentation is not on par with the extensive C++ documentation. The OpenMPI and IntelMPI bindings have differently structured namespaces and function names, making it necessary to write separate code depending on which implementation is used. Also, not all methods of the underlying library are wrapped. Last but not least, all Java bindings are marked as experimental. And while they have not changed in recent years, the vendors reserve the right to modify or remove them.

The Python performance in Figure 3.2 is the result of using the mpi4py library in combination with NumPy arrays. As with Java direct buffers, the achieved performance is on par with the C++ implementation. This is not surprising since

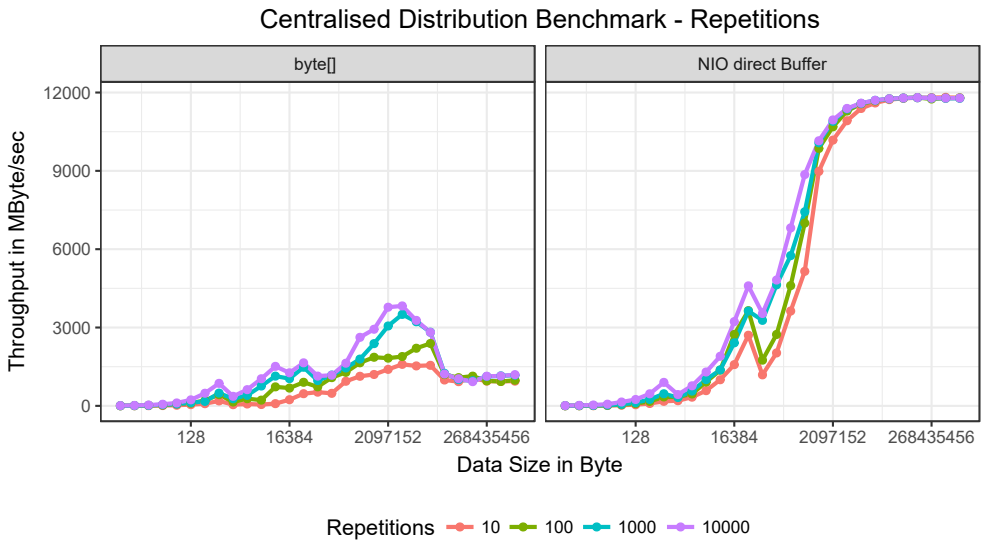


Figure 3.3: Varying repetitions for different interface functions with 8 nodes

NumPy arrays are also a continuous block of memory and can thus be directly passed to the underlying C++ library without any conversion [50]. As with Java, this performance drops considerably when using other data types that have to be serialised first [48].

There is a drop in performance at 250 MB in both the C++ and the Python, but not the Java implementation, in Figure 3.2. Since the Java implementation builds on top of C++, it should never be faster. Still, multiple measurements all came to the same result. There is no indication of additional logic inside the wrapper code, which would explain better performance. No obvious explanation could be found for this behaviour.

The performance measured in these benchmarks will degrade inside the real application as OPE will not use plain byte arrays but rather complex objects. These objects have to be converted into byte arrays and stored in direct buffers before being sent. The default mechanism for this is the Java serialisation, which is notoriously slow. However, performance can be improved by reusing already allocated buffers and writing custom serialisation.

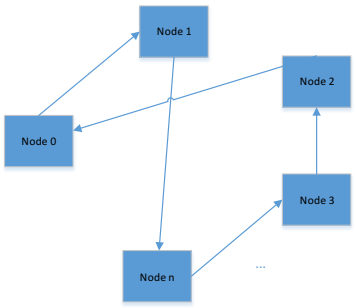


Figure 3.4: Benchmarking distributed data exchange

Conclusion

The results confirm the common assumption that larger chunks of data are transferred more efficiently. The usual cache effects are also visible. The achieved throughput is low and diverse enough to make transfer time a relevant factor when later designing the load balancer, especially for small data sizes.

Most importantly, the performance a Java or Python implementation can achieve is on par with the C++ implementation. This result confirms Java and Python are viable languages for the implementation of the system. However, additional overhead for object serialisation/deserialisation has to be measured and optimised separately.

3.3.2 Distributed Data Exchange Benchmark

After OPE executed a distributed calculation, the result are spread out over a number of nodes. If the next calculation is also allocated over various nodes, these results may need to be redistributed. Either because the calculation is computationally different, requiring a significantly higher or lower count of worker nodes; or the dimensional dependencies are different, making it necessary to restructure the data to meet those requirements.
To measure a baseline for this kind of transfer schema a micro benchmark outlined by the following steps (see Figure 3.4) is used:

1) Create n MPI worker nodes.

2) Each node will reserve x Byte of data. The assumption being that the previous calculation was evenly distributed. The generated results should thus also be uniformly distributed.

3) Each node will send the data to a random node. This is set up in a way, that each node only receives one piece of data. This implementation is essentially varying node permutations.

4) This sending is repeated multiple times and the time is measured.

Whereas the Centralised Distribution Benchmark measured the time only for the first node, here it is the slowest node that is measured. This happens by placing a barrier after all send and receive commands. The send and receive of each node are implemented as an asynchronous send, followed by a synchronous receive and finally a wait on the future of the send. This is the common implemenation for a ping-pong benchmark. One deviation from this pattern is the Java `Object` interface implementation since the OpenMPI Java bindings support asynchronous commands only in combination with NIO buffers. In this case, half of the nodes will first send data, and the other half first receives data.

As before, the variable that is varied is the size of the data. The permutations of node numbers that have been tested for both: been fixed for the complete benchmark and beeing changed for each repetition. The results did not change.

Results

The results of the benchmark can be found in Figure 3.5. As already seen in the Centralised Distribution Benchmark the results are very dependent on the used implementation. Again, the Java NIO and Python implementations are very close to the C++ reference.

Figure 3.6 shows the Java results for different repetition counts. Again, this is done to assess the influence of JIT compiler effects. In contrast to Figure 3.3, there is a speed increase in both implementations when using more than 100 repetitions. However, the effect of even more iterations is not as profound.

Somewhat unexpected is the steep drop in performance for data sizes starting with 256 MB. Given only this measurement it is difficult to explain the origins of this effect. It could be that the combination of simultaneous receiving and sending produces a problem with pinned memory, disabling Direct Memory Access (DMA). But without further tests, this assumption is pure speculation. Additionally, the technical specification sheets of the used network cards do not contain enough information for an adequate analysis. However, the drop is not as relevant for the conclusion of this benchmark. As it occurs with all the measured implementations,

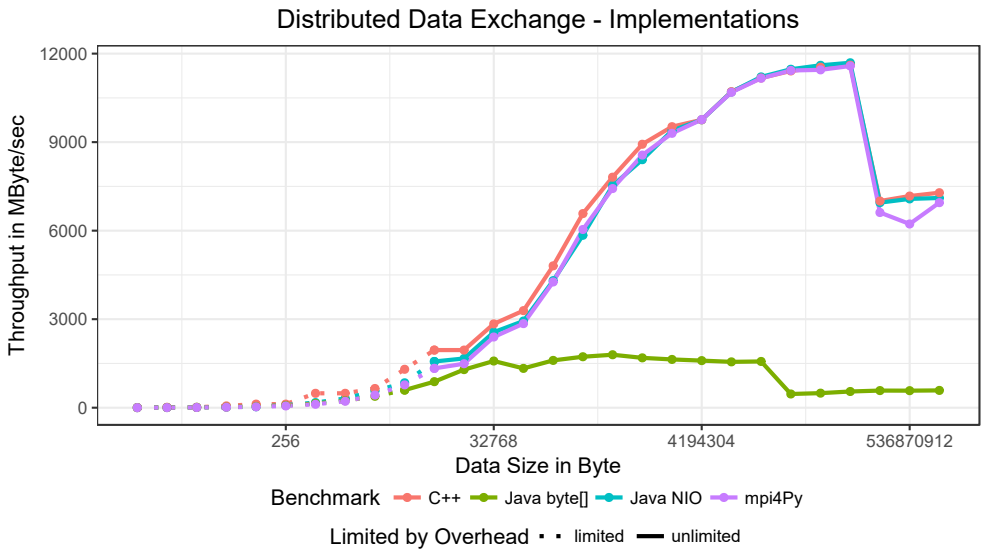


Figure 3.5: Different implementations of the distributed data exchange benchmark with 8 nodes and 1000 repetitions

even the reference C++ one, it can be assumed to be a technical problem on a lower level. As such, it does not influence the later design decisions.

Conclusion

The result of this benchmark confirm the previous results (see subsection 3.3.1): both Java and Python are viable languages when it comes to data transfer if the proper interfaces are used.

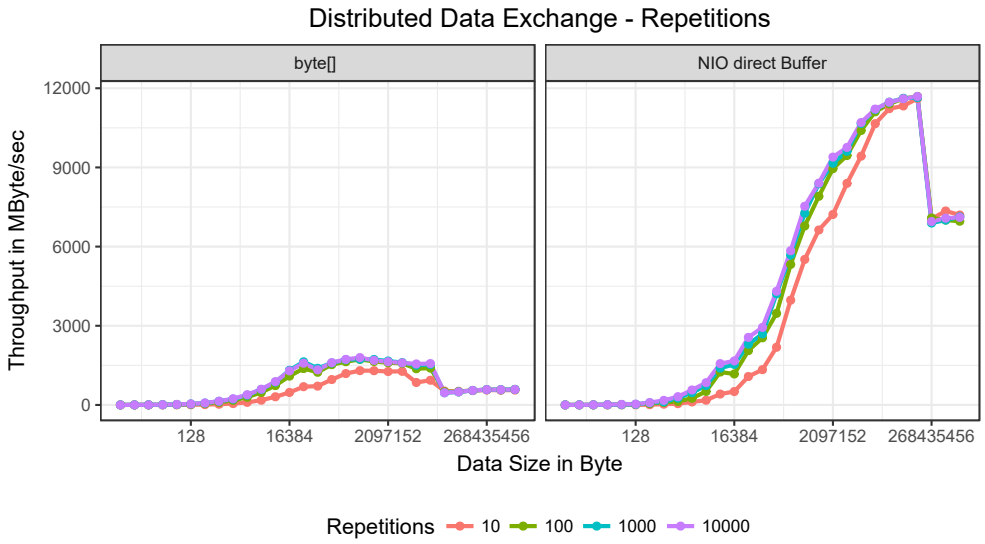


Figure 3.6: Varying repetitions for different interface functions with 8 nodes

4 Design

To understand OPE, it is necessary to understand its design: the parts it is made of and how they interact with each other. This chapter will discuss the steps, decisions and reasons behind those decisions that resulted in the final design of OPE.

In the broadest sense, the final design of OPE consists of two parts: The processing manager and the web interface. An overview of these components can be found in Figure 4.1 and should serve as a point of reference for the discussion in the following sections. The web interface includes the User Interface (UI) and the logic necessary to manage workflows. The processing manager handles the distributed execution of workflows.

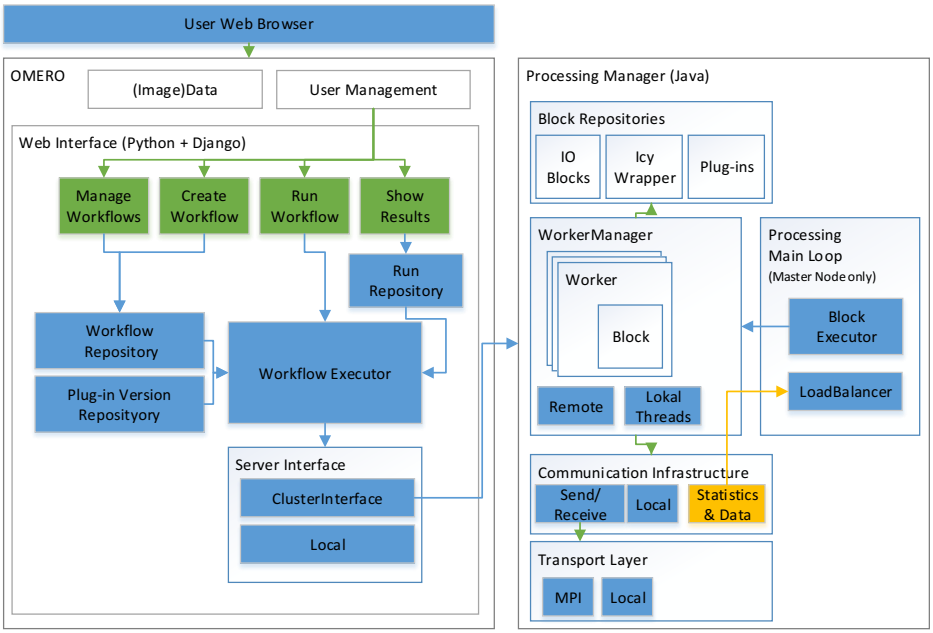


Figure 4.1: Architectural overview

4.1 Basic considerations

OPE is composed of two components: the Web Interface and the Processing Manager. Before explaining these modules in detail, this chapter illustrates the decisions that led to this structure. The three big goals of OPE are:

1) Offer a simple way for biologists to automate routine calculations. (Usability)

2) Ensure repeatability and reproducibility of these calculations. (Reproducibility)

3) Provide better performance through parallelisation. (Performance)

The first goal of usability dictates the requirements regarding the UI. Since the aim is to create an intuitive interface, a pure console application is not suitable. The use of a GUI results in a better usability for average users, specially if this GUI complies with the accepted iconography and usage patterns of the users domain. The goal can also be understood as a requirement for easy and flexible deployment. As a result, platform independent GUI technologies (such as Java AWT/JavaFX or C++ with QT) and web-based solutions are viable options.

The second goal can be met in a wide range of ways, ranging from writing custom workflow files to storing the data in some custom database or integrating everything in an existing system. Writing files is used by many workflow systems. The users of these systems can store their workflows inside a file for later reuse or protocol purposes. Given the input data and such a workflow file, the chances of being able to reproduce the calculation results are high. The only potential limiting factor is how precise these systems stores version information and having the correct versions of the involved tools at hand. While this level of reproducibility is a satisfactory baseline for what functions OPE should offer, the goal is to expand on this solution by also handling the versions problem, as well as automatically storing input, calculation workflow, and results inside the same system. Storing the workflow in a database has the same function. However, a database comes with a more limited expressiveness, depending on the used technology. Sharing and moving the data between databases can also be more challenging compared to using files. As an advantage, databases excel at searching and storing large amounts of data. They can also be used to enforce a coherent storage schema. One last possible solution is tightly integrating with an existing system, to achieve reproducibility. This solution defers problem onto the system that is being integrated with.

The third goal of using parallelisation for better performance originates in the realisation that many image calculation operations support easy parallelisation due

to their nature of requiring only local image information (see chapter 2). Since most modern computers have the capabilities to run operations in parallel, it should be a goal to make use of them.

The starting point for most considerations was the work that was done with the OMERO platform alongside the development of OPE as part of the DFG funded SFB 166 Receptor Light project. As a start, OMERO already offers excellent options for storing, displaying, and sharing biological images. The extension [51] done in the Receptor Light project further improves the capability to record all necessary information on how an experiment was performed, expanding the reproducibility tracking into the experiment. However, OMERO's features are rather rudimentary when it comes to the follow-up steps of running calculations on these images.

Addressing the overarching goals inside of OMERO is compelling. It allows the user to stay inside the same system when storing the raw data and the results, while automatically recording as many of the intermediate calculation steps as reasonably possible. For more information about the architecture and capabilities of OMERO see subsection 1.1.2.

As OMERO is primarily accessed through a web interface it already satisfies many of the formulated requirements:

- It is easy to use, as it follows standard web usage patterns.
- It does not require any specific deployment on the user side, as it runs in the browser.
- It already stores some of the reproducibility information.
- It offers a wide range of extension points, which allows the use of information stored inside of OMERO.

Because of all these factors, extending OMERO offers many benefits for the user. The added restrictions of having to comply with an extension interface are manageable, due to the flexible nature of the design and technologies used by OMERO. However, it does mandate the first technology choice: The user interface has to be realised as an OMERO web plug-in. The required technologies are Python and the Django web framework.

The next consideration is whether or not the whole system should be implemented inside this framework or if parts should be moved to a separate system. One of the substantial advantages of creating a single system in one programming language is

consistency. There is no need to convert data between languages, all system parts easily communicate with each other, and even if the system is separated into different sub-parts, they all use the same toolchain for development, build, and deployment. Since OMERO is already deployed on a server, this would require mere copying of the system into the appropriate OMERO plug-in folder (copy deployment). Additionally, if needed, all parts of the system can access OMERO. As OMERO has already ingested the image, this integration enables all plug-ins to directly access different image portions as well as metadata without the need to load a complete, potentially big, image. If user management is required, the extensions can also access the OMERO user management eliminating the need to implement a custom solution.

Most disadvantages of implementing everything inside an OMERO plug-in stem from a potential performance impact. OMERO web plug-ins are run inside the web service. The execution of a time-intensive computation would consequently impact the user experience for all users interacting with the website. OMERO itself solves this problem for its calculations by offloading them onto a separate calculation service, which can be scaled independently. As OMERO uses Ice to realise the interactions between its subsystems, this calculation service can also be written in a different language.

Using this calculation service would have been a viable option. However, such a solution requires extensive modifications of the OMERO code base. The result of which is a forked OMERO version, with all the usual update and maintenance problems such an approach incurs. The sensitive solution for this problem is to separate the user-facing aspects and calculation aspects into two separate subsystems and to deploy the calculation system on a separate machine. This division results in the web interface and processing manager modules that can be found in OPE.

Such a separation also offers more flexibility regarding the implementation of the calculation system. The only requirement is the accessibility through Python. The access could be implemented through everything form a CLI, a RESTful API, or RPCs. Following the intention of providing reasonable performance on top of usability and reproducibility, the calculation system should be able to utilise a compute cluster. This requirement is not as much a constrained, as it is a way to reduce the number of available technologies. Both the Ara Cluster and the Lofar Cluster support MPI, a widely used technology for distributing work over compute nodes. It supports many programming languages, such a C++, Java and Python. As shown in section 3.3, all three options offer a comparable performance when it comes to distributing data. This leaves two main factors for consideration:

Decision	Advantages	Disadvantages
separate GUI / Compute	flexible	2 languages/toolchains
GUI: Python + Django	easy to use integration with OMERO available libraries	not type save
Compute: use cluster	better performance	data transfer to cluster
Compute: use Java	high abstraction level available libraries platform independent	considered verbose
Compute: use MPI	industry standard fast	low abstraction level

Table 4.1: Advantages and disadvantages of design decision

calculation performance and programming complexity.

C++ is the usual first choice when it comes to computing performance. However, the impact of the computational performance can actually be decoupled from the performance of the overall computation system. By using the same techniques as the MPI wrappers, it is possible to integrate C++ code into a Java or Python application with little overhead. Therefore, it is reasonable to implement the control portions of the system using a managed language. As managed languages offer a more extensive range of language features, they promise a faster way to write code [52, 53, 54].

With the decision to use a managed language, there remain only two contenders for the programming language: Java and Python. Both offer a wide range of libraries, sophisticated Integrated Development Environments (IDEs), and a very active user base. Java does additionally enforce the use of Object-Oriented Programming (OOP) and strong typing. While both of these mechanisms potentially make the development slower, they ensure fewer bugs and easier maintenance [55]. In the end, since no hard requirement dictates the choice, Java was selected for its potential better maintenance behaviour as well as personal preference.

Table 4.1 summarises the implications of all design decision.

4.2 Web Interface

The web interface is the only part of the system that the user directly interacts with. OPEs processing capabilities can be used without this part of the system,

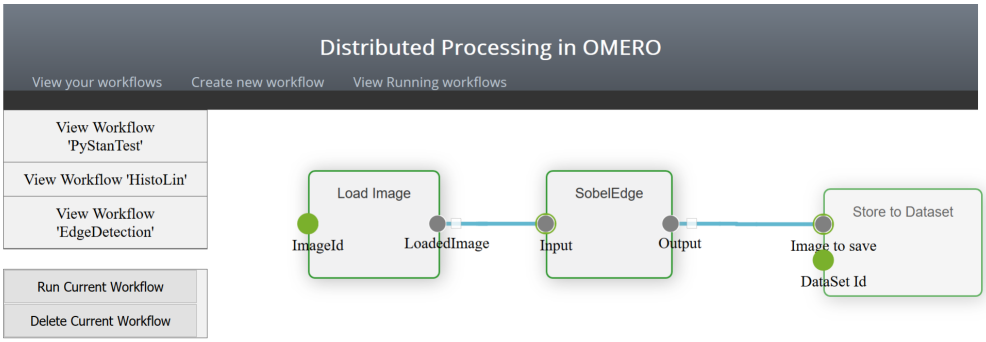


Figure 4.2: Screenshot of the web user interface for displaying workflows

but since the aim is to be user-friendly, it is necessary to offer a convincing UI. As discussed in section 4.1, the user interface of OPE is realised as a plug-in for the OMERO image storage system. Because of this dependency, the UI inevitably has to be based on Python and Django. On top of this base, common HTML extensions such as JavaScript can be used.

From a user's perspective, the web interface is used to design, manage, and start workflows. An image of how this system looks is depicted in Figure 4.2. Here, an already constructed workflow is displayed in the "View Workflows" page of the UI. From this page, the current workflow can be selected and consequently started or deleted.

The web interface has to satisfy two requirements: it has to be easy to use and it needs to act as a bridge between OMERO and the compute part of OPE. The user experience is achieved by using the same structure and iconography as used by many other websites. As an example, users expect navigation to be either at the top or left of the website, while current information is displayed on the right side [56]. An aspect of the user interaction with the web interface is the construction of workflows. This construction is done using graphical programming. As discussed in section 1.3 this approach enables users without programming experience to intuitively generate the desired workflow description. Implementing this graphical interface is made easy by a wide range of free visualisation libraries for JavaScript. After researching and evaluating some of the available possibilities, the JavaScript library jsPlumb is chosen to realise a blocks and connections based graphical programming interface. The reasons were its free license, comprehensive documentation, and adequate set of functions. All in all, four main websites are needed to interact with OPE:

- The “Create new Workflow” page is used to create a new workflow. By providing a selection of all the available blocks, the user can combine the desired operations into a complete workflow. Each block represents an operation and has areas (circles) representing inputs and outputs. Each connection links an output area of one block with an input area of another block and represents a data dependency.

- The “View Workflows” page is used to inspect already existing workflows. It is also be used to start them.

- The “Run Workflow” page is used to start the workflow. The user has to input all needed parameters as well as the desired versions for blocks under version control. When selecting images or datasets, the GUI assists the user by displaying the available options inside the OMERO structure.

- The “View Running Workflows” page is used to display an overview of the workflows the user has started, their status, as well as their results. If intermediates where recorded, a separate page will display the workflow and augment all connections with the intermediate results that represent these connections.

The bridge between OMERO and the compute part of OPE comes into play when a workflow is started. Such a start means transferring the needed data out of OMERO into OPE, triggering the actual calculation via the cluster, and finally the import of results into OMERO. These transfers are needed, as the used clusters can not directly access OMERO for security reasons. As a result, the processing manager was designed to work with files as inputs and outputs.

In addition to the described core functions, a number of augmenting functionality is performed by the web interface. Most prominently, the interaction with the version control system is handled by it. As is further discussed in section 4.4 and chapter 6, OPE derives most of its actual processing functionality from plug-ins. To accurately track the versions of these plug-ins and ensure reproducibility, they are placed under version control. To provide the execution of a specific workflow with the desired version of the plug-in, one of the parts of OPE has to interact with an external version control system. As the compute cluster does not have full network access because of security concerns, this interaction is also situated in the web interface.

To properly function, the web interface also has to store some additional data. There exist three datasets:

- The **block repository** contains all the work blocks a user can utilise to construct a workflow. This implies a partial redundancy since this information could also be retrieved from the processing manager. However, some work blocks should not be available to the user (benchmark and testing blocks), while others have no direct representation (load/store of data from disk). It would have been possible to offer an appropriate interface to access this additional information. But as it is essentially UI configuration information this option was rejected.

- The **workflow repository** stores all user-created workflows: what blocks are used and how are they connected. This does not include parameters the user is supposed to specify when starting the workflow, such as which image to process. The used version numbers are also not part of these descriptions.

  Currently, workflows are not access controlled. Since the web interface is part of OMERO, this feature could easily be implemented by accessing the OMERO user management.

  In the same manner as plug-ins, workflows could also be stored in a version control system. As such an implementation was postponed, as it poses some challenges in regards to user authentication between OMERO/OPE and the version control system. This decision does not negatively impact reproducibility, as each result is annotated with the complete workflow that created it.

- The **run repository** contains a list of all workflows that have been executed. This includes their ID, start time, status, and where the run description was stored.

  The actual run description is stored inside OMERO, alongside the results of the workflow. This description contains the complete workflow with all selected parameters, the OMERO IDs of all created results, run statistics, as well as the dataset inside of OMERO where all intermediates are stored (if the user requested them).

A complete list of the web interface features is given in Table 4.2. The more minute parts are better explained in the interaction between the web interface and the processing manager, which can be found in section 4.3.

- Offer a graphical programming environment for workflow creation
- Query source control for available plug-in versions
- Ensure the deployment environment
- Retrieve plug-ins from source control and compile
- Query OMERO for images, datasets, and other objects
- Export files from OMERO
- Deploy input data to the execution environment
- Start the execution of a workflow
- Monitor the workflow execution status
- Import results and statistics to OMERO

Table 4.2: List of web interface functionality

4.3 Processing Manager

The processing manager is the part of OPE, which will execute a workflow on a distributed compute environment. As discussed in section 4.1, the choices of programming language and technologies used in this part are somewhat flexible. The final decision was to use the managed language Java for its high flexibility, abstraction, and expressiveness. To interact with the distributed compute environment, MPI is used through a Java Wrapper provided by the underlying MPI implementation. This has no negative performance impact, as discussed in section 3.3.

The goal of the processing manager is to distribute the work contained in a workflow over multiple compute nodes. Additional requirements are the easy extension with additional functionality, enabling reproducibility, and following standard software engineering best practices and patterns.

Processing manager is realised as a master-worker architecture with a layer pattern. These two concepts will first be explained, discussing their advantages and disadvantages. Subsequently, the two most prominent technical aspects of the processing manager is presented in detail: the Scheduling and Load Balancing (see chapter 5) and the Plug-ins architecture (chapter 6).

4.3.1 Master-Worker Architecture

The Master-Worker architecture is a well-known pattern for general parallelisation [57] and can be found in a wide range of applications and abstractions. These range from language-internal constructs like the Java `ExecutorServices` to massively distributed systems like BOINC [58]. It is best practice for applications

involving the need to compute multiple subproblems that have few, clearly defined dependencies between them and can be calculated in parallel [59, 60].

The lately popular micro-service approach [61, 62] focuses on a similar kind of problem. However, it trades performance in favour of high reliability and maintainability. It also tends to rely on Representational State Transfer (REST), a architectural principle that demands that services are stateless and represents data as text based schemas such as XML or JSON. Because of this, it can be ineffective when large pieces of data have to be transferred.

A master-worker architecture consists of two parts: a master node and a worker node. The worker (or sometimes slave) node is responsible for handling the workload of the system. For OPE, this workload consists of running calculations on data. Beyond that, the worker node contains only minimal control logic. Because of this lower complexity, the worker node can be highly optimised and lightweight. In distributed systems there are usually multiple instances of the worker running at the same time, enabling scalable parallel computations.

The master node is responsible for distributing the work to the workers as well as administrative overhead. These tasks include:

- Analysing dependencies between the particular subproblems with the goal of achieving optimal performance of the system.
- Tracking the status of the subproblems and deciding what is executed at each given point in time. This includes considering possible dependencies between the problems.
- Managing the resources provided by the system.

This wide range of tasks makes the master node more complex in its design. Since it does not participate in the calculation itself, the master node can be implemented using more powerful, but potentially slower, high-level constructs.

If the system is not designed with mechanisms for fault tolerance, the master node is a single point of failure, meaning that its defect crashes the whole system. For now, the assumption is that OPE will not execute calculations, that run long enough to make fault tolerance a high priority requirement. As such, running redundant masters, which are synchronised with each other, is not a requirement. Neither is the use of check (or recovery) points [63, 64] during the calculation to recover results if a worker node crashes and its intermediate results are lost.

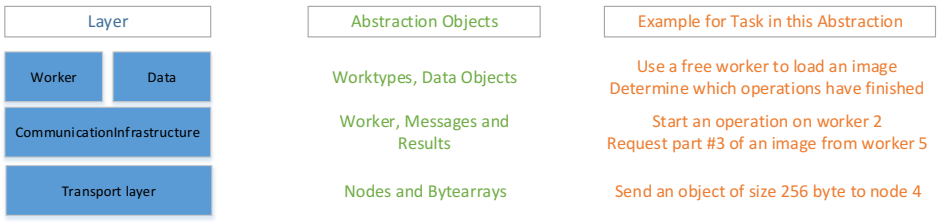


Figure 4.3: Layer overview

4.3.2 Layer Pattern

The layer pattern is a way to structure software as a set of software modules, called layers. These layers form a stack, in which each layer depends only on the next layer, often representing different levels of abstraction [59, 65]. OPE uses this approach of designing a software. It abstracts the communication and exchange of data between nodes by creating layers of different abstractions, each representing a granularity of exchange. The goal is to make the transfer of data between nodes transparent. An overview of the layers and their abstraction levels is given in Figure 4.3.

As a result, different calculation nodes are represented by a worker object in the top layer of OPE. These workers can be configured to run a calculation with very abstract input representations. Each worker is constructed and managed by a `WorkerManager`, a bridge pattern [66], that can be used to select from a range of communication models. The abstract input objects themselves are responsible for how their parts are distributed over the nodes and hide the logic to request missing parts. The worker implementation used in the production environment interacts based on messages, as it ultimately builds upon MPI. All other implementations employ variations of a local workers and are used for unit testing and non-distributed calculations.

The middle layer (or communication infrastructure) handles the communication between workers. It also serves as a data storage for the results. In the case of purely local processing via threads, this means all worker work on a shared memory model. In a distributed environment, the result of an operation is stored in the communication infrastructure of the executing node and a notification is sent to the master node for each such result upon its creation. The combination of all these notifications constitutes an abstract result, which is used for the next calculation. This distributed storage limits unnecessary data transfer. The desired middle layer

can be selected using dependency injection. A local communication infrastructure simulating distributed memory is also available and is used when performing unit tests.

At the lowest level, the interaction with MPI is implemented. Here, messages and data objects are converted to objects of the type `byte[]` and sent between nodes.

4.3.3 Master Node

The master node orchestrates the distribution of work over the worker nodes. The main logic for this management is situated in the `ProcessingManager` class. Inside this class, the workflow description is first converted into a work graph and initialised. After this initialisation, a main loop continuously executes the following steps. The loop is terminated if all blocks are marked as finished or an error occurs.

1) Find all work blocks which are ready to be executed. Ready means that all input ports of the block have been assigned a value and are marked as valid.

2) Get all available free workers.

3) Distribute the work onto the free workers
   a) Assign each block that is not parallelisable to a single free worker.
   b) If free workers remain, fill the workers by distributing parallelisable work.
      i. Estimate how big a work chunk needs to be to fill the desired work time (default 10 s).
      ii. Construct chunks and assign them to workers until either the complete input of the block is assigned, or no more free workers are available.
      iii. If free workers remain, handle the next parallelisable block.

4) Update the status of all blocks according to the results that have been received from worker nodes. This receive operation is handled asynchronously by the underlying communication infrastructure.
   Results do not need to be actual data, but can also be proxy objects for data that is remaining on the node which produced it.

5) Transfer data from the output of finished blocks to the next blocks according to the links in the workflow graph.

6) [Optional] Instruct workers to store data that no work block depends on any longer to disk. This step can be activated to keep the memory footprint of the workers low.

As discussed before, on this level the distributed nature of the system is abstracted away by the workers, which are managed inside an `IWorkerManager`. Depending on the actual implementation, such an `IWorkerManager` could provide workers that work locally as threads or proxies which encapsulate the logic to offload work to different compute nodes in a cluster.

The main loop and the involved classes and interfaces are displayed in Figure 4.4. On the left side, the described main loop is depicted again. On the right side, the involved interfaces and classes can be found. Situated at the top is the `IWorker-Manager` interface, that is used by the `ProcessingManager` to retrieve workers which implement the `IWorker` interface. There exist three implementations of `IWorker-Manager`: `MainThreadWorkManager` implements both `IWorkerManager` and `IWorker` as it will run scheduled work in a blocking manner on the calling thread, which will most likely be the main thread of the application. `ThreadsWorkerManager` will execute work on locally running threads. This implementation does not rely on a communication infrastructure. Instead, it stores results directly in the work graph. Last but not least, `RemoteWorkerManager` will use a communication infrastructure to distribute work (potentially) over multiple nodes. The actual form of the distribution depends on the used underlying `ITransportLayer`. In any case, using a `RemoteWorkerManager` requires a `RemoteWorker`, which will handle the calculation, and a `RemoteWorkerMasterSide`, which represents the worker to the master nodes and encapsules all necessary communication.

`ICommunicationInfrastructure` has only one implementation: `SendReceiveInfrastructure`. It implements the communication between distributed nodes using messages. No other implementation was provided, as MPI is message based and is the only currently supported underlying technology. Finally, `ITransportLayer` is implemented by two classes: `OpenMpiTransportLayer` represents a message exchange through MPI and is used in the cluster environment. `LocalTransportLayer` is using local message queues which exchange messages. The local transport also includes object serialisation and de-serialisation, which is inherently present in all distributed implementations and a common source for implementation problems.

Another abstraction on the level of the `ProcessingManager` has to do with handling the distributed data objects, such as images and tables. Without going into

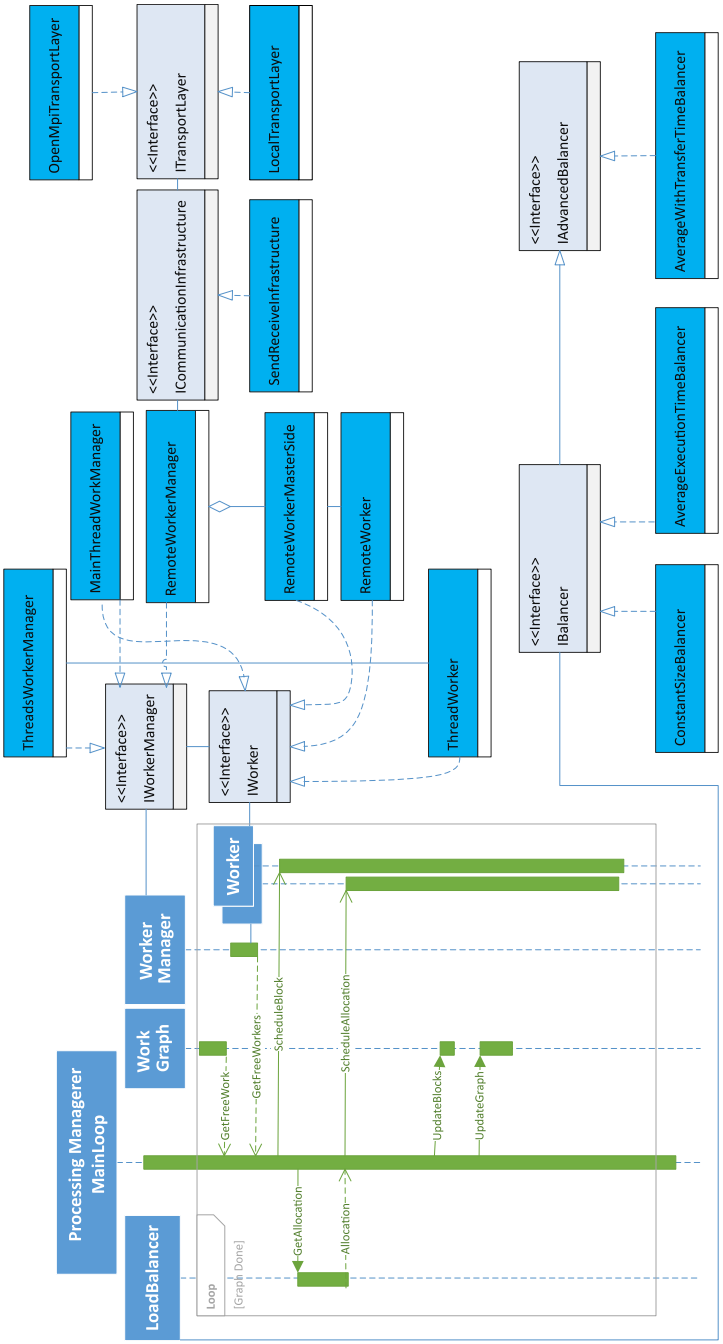


Figure 4.4: Overview of the masternodes main loop and the involved interfaces and classes

too much detail, for each splittable object, a proxy object has to be created which handles the local and distributed parts of the object. Such a proxy can then be used by a worker to request parts or all of the data. Internally, the proxy will use the communication infrastructure to acquire local parts and request distributed parts and sub-parts, when needed. With the usage of these proxies, image, and tabular data is also moved out of the work graph into the communication infrastructure. This shift enforces a more explicit separation between execution and data management. It also means that when a work allocation is sent to a worker, only a proxy is used for the data, which significantly decreases message size and defers actual data transfer to a later time point and a somewhat decentralised schema.

On a final note, one non-trivial problem is how to determine when an operation is done. As long as only images are considered and the used transformation is sufficiently formalised it can be determined how big the resulting image is and thus the operation can be considered done if the predicted amount of data was generated. With the introduction of projection operations and tables, this decision becomes harder to make. As a solution, OPE tracks the progress of an operation by how much of the input data has already been requested, instead of tracking how much output has been created.

4.3.4 Unit Testing

Unit and system tests are used to ensure that OPE performs as expected and that changes do not break existing correct behaviour. These tests will check specific classes regarding their correctness by presenting class methods with parameters and checking the results against known expected results.

System tests, on the other hand, can test the execution of complete workflows. These tests are realised by replacing lower abstraction levels with locally executable implementations or specialised test classes. There are four levels of system tests used in OPE. These levels are explained in order of the number of components involved.

At the simplest level, `MainThreadWorkManager` is used to run all computations on the main processing loops thread in a blocking manner. Since only one thread is involved in this test, no timing issues can be examined. However, the main loops basic logic, as well as program startup and shutdown are covered. As no concurrent operations occur in this test, the expected behaviour is easy to predict and does not change between test executions. This behaviour makes it ideal to test and debug basic functions.

On the next level, the `ThreadsWorkerManager` can be used to run multiple calculations simultaneously. In such a test, all intermediate results are stored inside the work graph, as opposed to the communication infrastructure, which is the case for the production system. As a result, no remote parts have to be requested and communicated. This test setup is ideal for testing that work is correctly distributed onto multiple workers as well as checking for deadlocks in the interaction of multiple workers.

As a next option, `RemoteWorkerManager` can be used in conjunction with `LocalTransportLayer`. This configuration is very close to the actual deployed version. `LocalTransportLayer` simulates the sending and receiving of messages in the same manner as is done when using `OpenMpiTransportLayer`. This includes serialising and deserialising of messages. This serialisation is qualitatively new, as the system tests mentioned until now can pass results by reference, whereas the complete system clones data when it sends them over the network. It is thus necessary to cover both mechanisms in tests, as the difference in behaviour between a clone and a reference is substantial. Using `RemoteWorkerManager` also adds the abstraction layer of using an `ICommunicationInfrastructure`. Among other things, this abstraction moves the storage of intermediate results from the work graph to the communication infrastructure, representing the results in the graph only by proxy objects. As hinted at before, these proxy objects handle the request of remote and local data parts. Using them in this system test ensures that these interactions are also covered by tests.

The last and most realistic level of a system test would be to execute an actual workflow on a cluster and compare the processed result with an expected result. As such an approach is hard to debug and requires a more complex toolchain, it was not implemented.

Overall, making low abstractions of the system replaceable for testing ensures that the system will behave the same way in a test and the real application. Conversely, this high similarity means that when an edge case workflow fails after a change, it can be easily converted to a system test. The test is than debugged and the problem fixed. The test case, which was created from the workflow, can then stay behind as part of the test suite, further improving the code covered.

As a result of the presented approach, OPE reaches a code coverage of 96% for all classes, 87% of the methods and 81% of the lines of code, as measured by the IntelliJ coverage tool. The not covered parts are either part of the low-level MPI access, which cannot be tested locally, example work blocks, that are not used as

they perform trivial operations, or safety checks, that are not strictly necessary but are in place regardless.

4.4 Reproducibility considerations

The goal of reproducibility in the context of microscopy image processing is to enable a scientist to determine how a result was created, including all the involved steps. As an example, if the final result of an experiment is a graph depicting the number and size of cells changing over time, reproducibility refers to the ability to determine all steps from the sample preparation to the final graph. The following partial list enumerates questions concerning the in-between steps and the information, data, or files needed to make these steps reproducible. The list is in reverse order, taking the final graph as a starting point.

8) How was the graph created. This can be a script, a program, or an Excel sheet.

7) What was the input data for the graph generation. This is most likely a data table containing the result of analysing cells in multiple images.

6) From what image was each point in the graph generated. This can be non-trivial, as each point refers to a row in the input table, which resulted from an image processed in a previous step.

5) How were the cells in these input images measured. This includes the used tool, in which version, and with what parameters.

4) If and how the individual images where preprocessed before the cells where analysed. This could be a script, a program, or by hand.

3) Where is the raw data stored. This can be a network share, an external hard disk, or an image storage system.

2) How was the raw data created. Using which measurement instrument, with what settings, and when.

1) How was the measured sample created. This can be information on how a organism was modified, what dyes where used, and what sample preparation steps where performed.

In the context of this work, the solution chosen to ensure provenance recording in all steps is the combination of OMERO and OMERO plug-ins. Storing information on step 1) is the domain of the lab notebook, step 2) is handled by the measurement instrument, and step 3) is the core functionality of OMERO. All three steps can be bridged by storing the image and the information traditionally kept in the lab book in OMERO. This option was implemented during the DFG funded Receptor Ligh project. The result is the CollAborative Environment for Scientific Analysis with Reproducibility (CAESAR) extension for OMERO [51, 67, 68, 69]. CAESAR stores information about the experiment alongside the generated raw data inside of OMERO. CAESAR uses input forms to record the experiment information in a structured manner. These forms were created in collaboration with biologists and are designed to be both complete and general enough to cover a wide range of biological applications. Additionally, a standardised file organisation schema is enforced as experiments are mapped to OMERO datasets.

Steps 4) to 8) represent processing steps (preprocessing, analysis, and visualisation of the result). Traditionally, they are performed by the wide range of software solutions introduced before (see chapter 2). Most of these solution are not integrated into OMERO, or other storage solutions for that matter. Therefore, it falls to the user to record the reproducibility information. To remedy this problem, OPE will automatically record the necessary information when executing a workflow. Two aspects need to be addressed to accomplish a complete reproducibility record. First, the complete preservation of the processing workflow, used versions, and data is needed. Second, the storage of this information has to be considered.

Information needed for Reproducibility

As stated before, the minimum information required to ensure the reproducibility is the workflow, the used versions, as well as input and output data. The options to record these three pieces are evaluated next. As OPE uses a graphical programming approach, the workflow can be formalised into a description by recoding which blocks are used and how these blocks are connected. It is common to represent this description in a human readable format, like XML or JSON. As a workflow description can be stored inside the workflow system, the decision has to be made if the description is duplicated or referenced when compiling the reproducibility information.

In addition to the workflow description, the complete version information has

to be recorded. As OPE, and many other tools, rely on plug-ins to extend their capability, the complete version information of these plug-ins is also required. It is sufficient to record version numbers to achieve this goal, if a consisted version management is used. In practice, such a version management often involves plug-in-repositories, in which programmers have to commit the plug-ins. OPE uses a less rigid but functionally equivalent approach. Each plug-in is stored inside version control repository, such as Git. When starting a workflow, the user picks the desired commit hash of the plug-in. As version control systems offer the possibility to augment commits with version numbers or labels, this approach includes a version management without additional overhead.

Preserving information about the input and output data is the most challenging part of reproducibility, as file names and paths are ambiguous and can change. Alternatively, using file IDs provided by a storage system is more convenient. However, IDs lose their meaning if the storage system can not be accessed. Another common approach to verify the file is to generate a check sum using a hash function. While it is expensive to search a file based on it's hash, it is a common approach to ensure that a potential candidate is indeed the desired file.

When collecting the reproducibility information, OPE will record as much information as possible. Even though the resulting description takes up more space, this approach ensures that the maximum amount of information is contained, even when studying the description without access to OPE. The recorded information is as follows:

- A copy of the original workflow description. This includes: the used blocks, the connections between these blocks, as well as complementary data on how to display the workflow.
- The full information on each block. This includes: the name, inputs and outputs, the type, and source control information like repository and commit hash.
- Information on the input and output files. This includes: file name, OMERO ID, size, and the files SHA-256 hash.
- The OMERO ID dataset in which the intermediate results will be stored. These intermediate results are only stored, when desired by the user.

Storage of Reproducibility Data

The gathered reproducibility information has to be stored. In many systems, all decisions regarding this storage are delegated to the user. To remove this additional administrative overhead, OPE automatically stores this data inside OMERO. As the reproducibility data is no image, storing it inside an OMERO dataset is not possible. Four general solution to resolve this challenge are conceivable:

1) Store the reproducibility data in a file and attach this file to another OMERO object. In most cases, this object will be the workflow result.

2) Use other attachment options, such as tables or key-value pairs, to store the data inside the database that OMERO uses.

3) Write a dedicated plug-in for the storage. Such a plug-in will maintain its own database.

4) Extend OMERO to handle the reproducibility information natively.

The solution selected for OPE is number 1), as it requires little additional implementation and a user can easily download the complete data. Solution 2 has the advantage of making the reproducibility data accessible for the OMERO internal search. However, it is difficult to extract the data from OMERO if needed. It is viable to implement solution 2) in addition to solution 1), if the search integration becomes desirable.

Lastly, solutions 3) and 4) require an unreasonable amount of additional implementation and maintenance effort. They where thus rejected.

5 Scheduling and Load Balancing

Distributed systems inherently have to decide how the work is distributed among its parts. To solve this problem, a range of scheduling [70, 71] and load balancing [72] techniques exists. To select an appropriate schema, it is necessary to define what information is known and at what time.

5.1 Problem Properties

To evaluate the scheduling problem, it is necessary to analyse the problem properties in regards to the factors of task cost, task dependencies, and task placement. As the computation is given as a workflow description to the OPE, a great deal of information is known at the start of the execution. Also, the workflow does not change during the execution. However, operations define rules for decomposing a problem into independent sub-problems. As a result, OPE can flexibly structure problems into sub-problems using these rule, making the used schedule more flexible.

Cost Each workflow consists of a number of operations. As most of the operations are provided by means of plug-in, not all information about their cost is known in advance. The possibility of multiple versions of the same plug-in makes it also impractical to gather runtime statistics on all possible operations. The possibility to perform a small benchmark on the operation to estimate the concrete costs for a given plug-in and version was also rejected, as some operations have input-dependent runtimes. This makes it difficult to construct viable micro-benchmarks at runtime.

As a way to still gather some cost data, OPE will gather runtime statistics on each given plug-in during the execution, ignoring the version. This approach will allow OPE a rough cost estimation at the beginning of a workflow execution, based on the execution of previous workflows. This estimation will improve as more and more pieces of data are processed in the workflow. This cost estimation will also adapt to content dependent operations, leading to better estimations.

Communication is another cost factor that has to be considered. As network structure does not change often, it is possible to sufficiently measure it once and use these measurements from that point on. However, as the infrastructure to measure costs at runtime had to be implemented anyway, it will also be used to measure communication cost at runtime.

As a consequence, it is possible at runtime to estimate the cost of calculation and communication reasonably before executing the workflow. However, a precise estimation is not possible. This restriction most noticeably limits the option to determine the remaining time a running operation requires.

Dependencies All dependencies between operations are known at the start of a calculation, as the workflow does not change after it has been submitted. However, this statement is only true on a high abstraction level. The parallelisation of work blocks means that an operation is decomposed into sub-operations at runtime. As this decomposition is dependent on factors such as data content, image size, and plug-in runtime, it cannot be sufficiently planned in advance.

Furthermore, the size of an operations result image cannot be determined. This limitation is due to low requirements concerning what information a programmer has to provide when writing a plug-in. Furthermore, the size of the result is content dependent of some operations. This design makes plug-ins more flexible.

As the rules for decomposing an input are known for each operation, it is possible to identify groups of operations that can be executed in parallel before the start of the execution. However, the number of actual sub-problems is only determinable at runtime. This number is also flexible within the limitation provided by the plug-ins parallelisation rules.

Placement Currently, the processing manager is run on clusters, which are structured into homogeneous groups. As such, there exists neither hardware nor software reason to differentiate between workers. The number of involved nodes is also fixed, as the obligatory use of a resource manager requires a job to specify the number of required nodes.

At runtime, it is very easy to determine which operation is currently executed on which node. The same is true for the storage locations of intermediate results.

A summary of the problem properties can also be found in Table 5.1.

Factor	Information	Time of Knowledge
Cost	rough estimation model for operation runtime	known at start
Cost	rough estimation model for transfer time	known at start
Cost	precise operation runtime	not known at start
Dependencies	dependencies between operations	known at start
Dependencies	input decompositions possibility	known at start
Dependencies	size of intermediate results	not known at start
Placement	number of worker nodes	known at start
Placement	number of busy and free worker nodes	known at runtime
Placement	schedule of intermediate nodes	known at runtime

Table 5.1: Scheduling problem properties

5.2 Scheduling Considerations

It is possible to narrow down the scheduling options given the previously discussed problem properties by following the schema outlined by Casavant [70].

Local vs. Global As OPE handles the distribution over multiple nodes, the scheduling problem is a global one. The local scheduling on the individual nodes is left to the individual plug-ins, assuming that they reasonably use the provided resources.

Static vs. Dynamic Scheduling cannot be done statically in advance, as the cost of the individual operations is not completely known in advance. This limiation is both due to the rough time estimates as well as the unknown number of sub-problems.

Distributed vs. Non-Distributed For OPE, the choice of whether or not to distribute the scheduling logic is a question of complexity. Centralising the scheduling in a non-distributed way simplifies the involved algorithms. Furthermore, a centralised solution helps to separate calculation from organisational logic, especially in combination with a Master-Worker Architecture. The fault tolerance inherent in many distributed algorithms is also not required. Finally, the complexity of a schedule is low as the considered workflows are not very big, as stated in the initial assumptions (see section 2.6).

Solution Ultimately, the presented scheduling problem is solved by dynamically scheduling work in a non-distributed fashion. This scheduling is done via a greedy algorithm. Whenever a worker node finishes an operation, the algorithm tries to fill

all free worker nodes with the next operations for which all operations they depend on have finished. To counteract some of the draw backs of greedy algorithms, OPE will not schedule all the work onto the free workers. Rather, the time estimation is used in conjunction with the operations' decomposition rule to construct sub-problems which take roughly ten seconds to complete. While this may not be the most sophisticated algorithm, it is sufficient as a starting point. The logic employed by the greedy algorithm to evaluate potential solutions is expressed by the load balancers that are discussed in the rest of this chapter.

5.3 Load Balancers

The used load balancer is the dominant factor when deciding how the workload is distributed. This section discusses all the used load balancers, covering the assumptions about the system they make, the advantages and disadvantages of the respective approach, as well as the benchmarks used to evaluate their performance.

All balancers try to estimate a subset of a given piece of data, that will fill a given time span with a given calculation. The rules underlying the creation of these subsets are provided by the operations/plug-ins. Since OPE deals with image data, this estimation translates to a pixel count, in the more straightforward cases, and estimating image areas in the more advanced cases.

All load balancers were run through a series of benchmarks to evaluate their performance in comparison with each other. To make these benchmarks more general and remove input specific artefacts they will use no actual calculation block, but rather dedicated delay blocks. These delay blocks will pause the execution for a calculated amount of time, before passing the input along as the output. Different models of how the delay time is calculated are possible, allowing the representation of $\mathcal{O}(1)$, $\mathcal{O}(n)$, $\mathcal{O}(n \log n)$, ... algorithms.

To keep the analysis as simple as possible, the benchmark will only consist of three blocks: Input $\longrightarrow$ Delay $\longrightarrow$ Output. The image is loaded onto a single node, is then distributed across multiple nodes for the delay block (representing a calculation), and is collected again on a single node for the output. A more complex workflow is explored in chapter 7.

The method used to split the image into sub-images has a performance impact, as it dictates how efficient memory operations are performed when splitting and merging the parts. For the following benchmarks, the delay block is configured to have no dimensional requirements and will use the most efficient splitting OPE

supports.

To evaluate different aspects, one parameter is varied while keeping all others constant. The parameters that can be varied are:

- The *problem size* represents the size of the input image. When changing it, the expected change in *runtime* is an increase in the same fashion as time behaviour of the delay block modelling the algorithm. For example, a linear increase in the case of the linear delay block.

  However, if the *problem size* is too small, not all available compute nodes are used. A bigger *problem size* could thus lead to better utilisation of the available resources, producing a better than linear behaviour.

- The *workfactor* models the computational intensity of the used operation. In the case of the used linear delay block, it is a multiplicative factor to the pixel count. The main reason for introducing this parameter is that it can be easily changed in benchmarks. In contrast, changing the pixel size would necessitate generating a new image. The expected influence of a change is consequently the same as for the *problem size*.

- The *delayfactor* is used to reduce the network performance artificially by slowing down data transfer.

After choosing which parameters to vary and executing the benchmark, the resulting data is evaluated. Overall runtime is an obvious factor to consider. To better understand how this overall time is composed, OPE tracks the time for certain parts of the operations. The diagrams shown later break down these components over all involved nodes. The individual components are:

- *Initialisation* captures all work that is done to initialise work blocks before executing the desired work types. For internal work blocks like loading, storing and benchmarking this time is very small. These components primary purpose is to track the time needed to load work blocks through the plug-in infrastructure. For example, the Icy block repository needs a one-time initialisation. This initialisation takes in the order of a few seconds and incurs when a worker accesses an Icy block for the first time.

- *Input Preparation* refers to the gathering and the preparation of input data. This effort includes retrieving object parts from other nodes. It also includes

the conversions needed for external plug-ins that use their own types to represent images.

- *Input* and *Output* mean the reading of the input image as well as the writing of the result image. The input includes initialisation overhead for the used Bio-Formats package, which does not incur a second time when writing the output.

- *Calculation* is the time spent in the delay block, thus simulating a calculation.

The overall time needed to complete the calculation is measured to assess the performance. Additionally, as a way to easily judge the utilisation of the system, the percentage of time spent in the calculation is compared to the overall time. The resulting value is between $[0, nodes-1]$, which is displayed as percent in the following figures. The maximum value of $(nodes-1) \cdot 100\%$ is reached if all worker nodes are running only calculations for the complete duration of the calculation. The "-1" stems from the fact that the master node does not execute any calculation.

To keep the results comparable, each benchmark was run at least 100 times, calculating the average of all results. Each run was executed on four nodes on the ARA cluster, resulting in three active worker nodes and one master node. The input image size was fixed to 100x100 pixels, as the *workfactor* was used to vary the amount of calculation required to complete the workflow. Later, the *delayfactor* will also be varied to evaluate the influence of the network.

5.4 Constant Time Balancer

The simplest model to estimate a number is a fixed value. Such a balancer implicitly assumes that the calculation time is constant and ignores all other factors. The underlining model represents an $\mathcal{O}(1)$ algorithm in which all other effects, like data transfer and block initialisation, are negligible. The resulting logic has the advantage of requiring only one parameter: the slice size. Since this parameter is part of the system's configuration, the need to measure and retain runtime statistics is also eliminated.

Figure 5.1 shows the runtime results for different *workfactor*s using a constant time balancer. The upper portion of the figure shows times spent on different aspects of the workflow, while the lower portion of the figure shows the time percentage of the

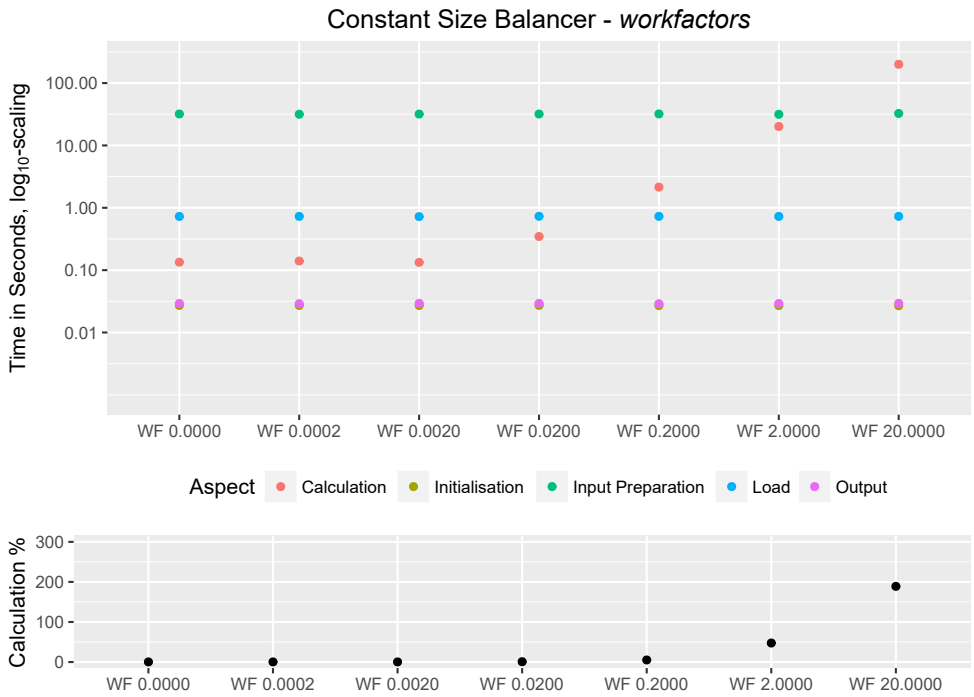


Figure 5.1: Benchmark of the constant time balancer for different *workfactor*s

time spent executing the calculation. As four nodes are used for the benchmark, the best possible result in this portion is 300%.

The slice size in this benchmark was set to 100 and a 100x100 pixel image was used as the input. The only component of the overall time that is changing between the different *workfactor*s is the time spent in the calculation. This is to be expected, as regardless of *workfactor* both how the calculation is split and how it is distributed over the available nodes does not change.

The different *workfactor*s are chosen to differ in factors of ten. To acknowledge this, the Y-axis is scaled on a log_{10} basis. As expected in this scaling, the amount of time spent on *Calculation* for *workfactor*s 0.02, 0.2, 2, and 20 increases by the same portion. Meaning that ten times the time was spent on this aspect of the calculation. The measurements of the remaining lower *workfactor*s can be assumed to be dominated by overhead and do thus not share this increase.

Besides the *Calculation* time, the overall time is dominated by the time for the *Input Preparation*, which includes data transfer. In contrast, general *Initialisation*

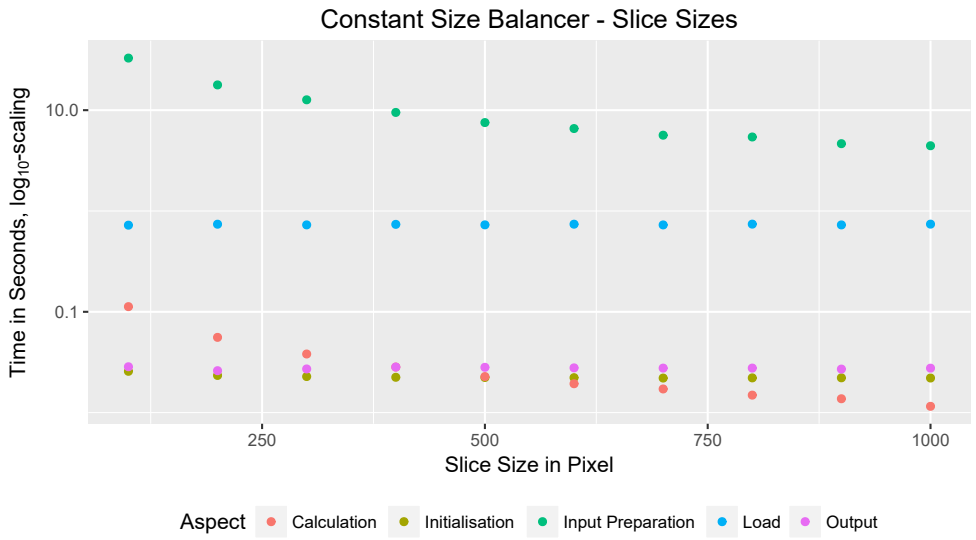


Figure 5.2: Benchmark of the constant time balancer for different slice sizes and a *workfactor* of 0.

as well as *Input* and *Output* of data is negligible. It can be assumed that input preparation is such a dominant factor because the transfer of the small data slices (100 pixels) is inefficient. This assumption fits the results of the transfer benchmarks (see section 3.3).

To further examine this assumption, Figure 5.2 shows the influence of different slice sizes at a constant *workfactor* of 0. As the slice size drops, so does the *Input Preparation* aspect of the overall runtime, since the number of necessary data transfers drops in the same manner. The noticeable drop in *Calculation Time* can be attributed to fewer calls to the calculation routine since the actually waited time is 0 and only measurement overhead is measured. A last point hinting at the non-optimal performance is the percentage of time spent in that calculation as shown in Figure 5.1. The results show that the balancer does not use the available resources very efficiently, given that the best possible value for this calculation is 300%.

Conclusion

As expected, using a constant slice size to decide the workload of the nodes yields a suboptimal result. The resulting system behaviour spends too much time on communicating data and is not flexible enough to adapt to more complex workflows.

However, it can be used as a tool to debug the system, as it produces a predictable work distribution.

To improve its performance, the used slice size needs to be tuned in accordance with the input data and the workflow. This could be a viable option when the goal is to remove load balancing from the execution and delegate it to an outside manager.

5.5 Average Time Balancer

A simple improvement over the Constant Time Balancer is to measure the actual time needed for a given work type and use these measurements to estimate the time needed for arbitrary data sizes. The underlying assumption for this estimation is that calculation time is only dependent on data size and not on the data itself.

The function to describe the relationship between data size and calculation time can be made easily exchangeable depending on the calculation type. As the used delay block implements a linear fashion the estimation in this benchmark will use linear regression on the available time measurements to calculate its result.

The performance of this balancer is shown in Figure 5.3. When looking at the composition of the overall time two things stand out: The time spent in *Input Preparation* increases for bigger *workfactor*s. Also, the time spent on *Calculation* is significantly lower for small *workfactor*s than when using the Constant Time Balancer.

The reason lies in the fact that the Average Time Balancer will schedule the complete calculation on a single node for small *workfactor*s, minimising the overhead. Only with bigger *workfactor*s, additional nodes will participate in the calculation until all nodes are utilised for the biggest *workfactor*.

When looking at the percentage of time spent in calculation, it is apparent that the resource usage is significantly better than for Constant Time Balancer. Particularly when considering the result for *workfactor* 20. In this benchmark, both balancers will use all available nodes, as often as they can. Consequently, the better ratio for this balancer must stem from a better, in this case bigger, data size and thus better network utilisation.

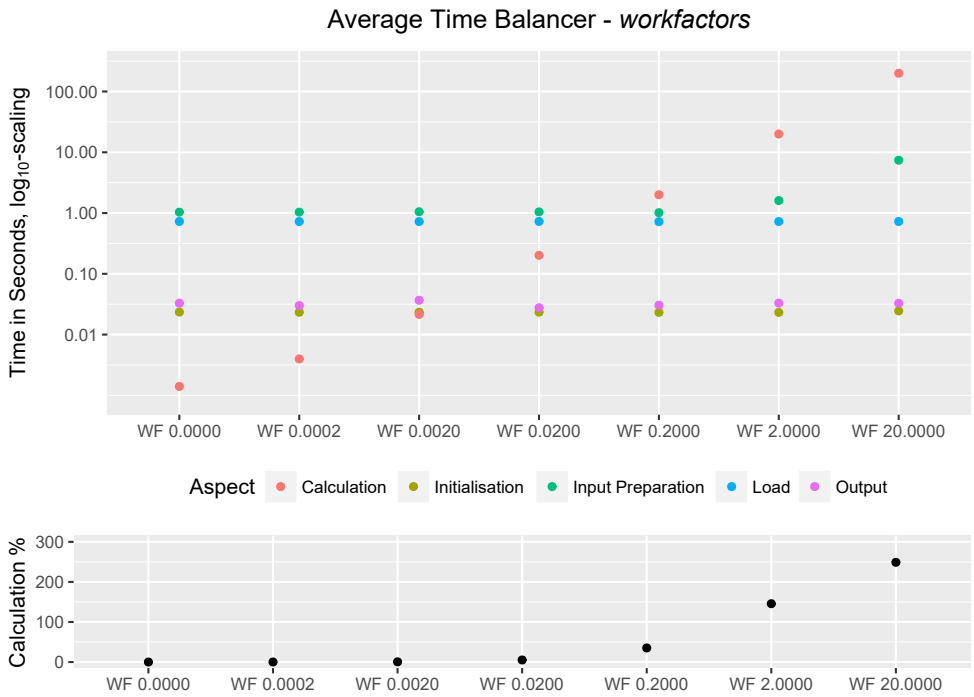


Figure 5.3: Benchmark of the average time balancer for different *workfactor*s

Conclusion

The Average Time Balancer is a considerable improvement over the Constant Time Balancer. To achieve this, OPE needs to track the needed execution time for each type of calculation and persist these statistics between runs.

5.6 Data Transfer Balancer

An improvement of the Average Time Balancer is to consider also the costs of transferring data between nodes. However, this requires considerably more information about the state of the system than the previous balancers. On top of the execution time statistics for the different work block types, information on the data transfer costs is also needed. The balancer also has to know the current state of the system, the position of the parts of data, and for what node the estimation is being calculated.

This increased need for information of the balancer requires a more complex

interface compared to the more simple balancers. This increase in complexity is also found in Figure 4.4, where the `IAdvancedBalancer` interface is introduced to provide the required functionality.

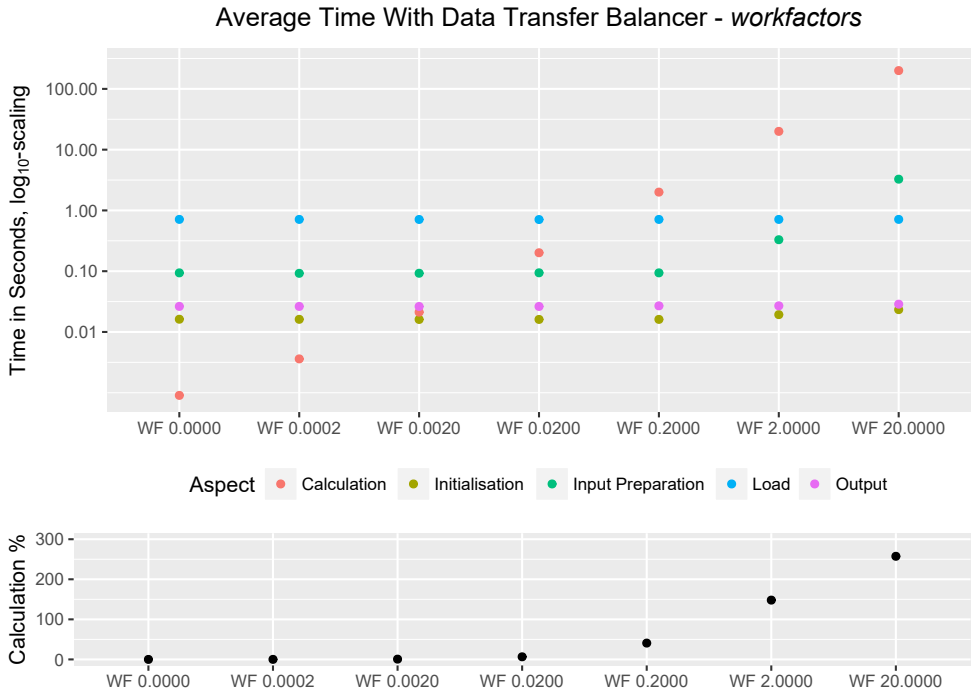


Figure 5.4: Benchmark of the data transfer balancer for different *workfactor*s

The results for the Data Transfer Balancer are shown in Figure 5.4. The most prominent difference compared to the previous benchmarks is that the time spent in *Input Preparation* is the smallest of all balancers. Besides this, the results are comparable to those of the Average Time Balancer.

The most likely reason for this similar performance is the comparatively low cost of sending data, outside of the overhead limited regime. As both balancers construct few and big data chunks, the cost to transfer them has a small impact on the overall calculation time. This will further be investigated in a further benchmark comparing all three balancers (see section 5.7).

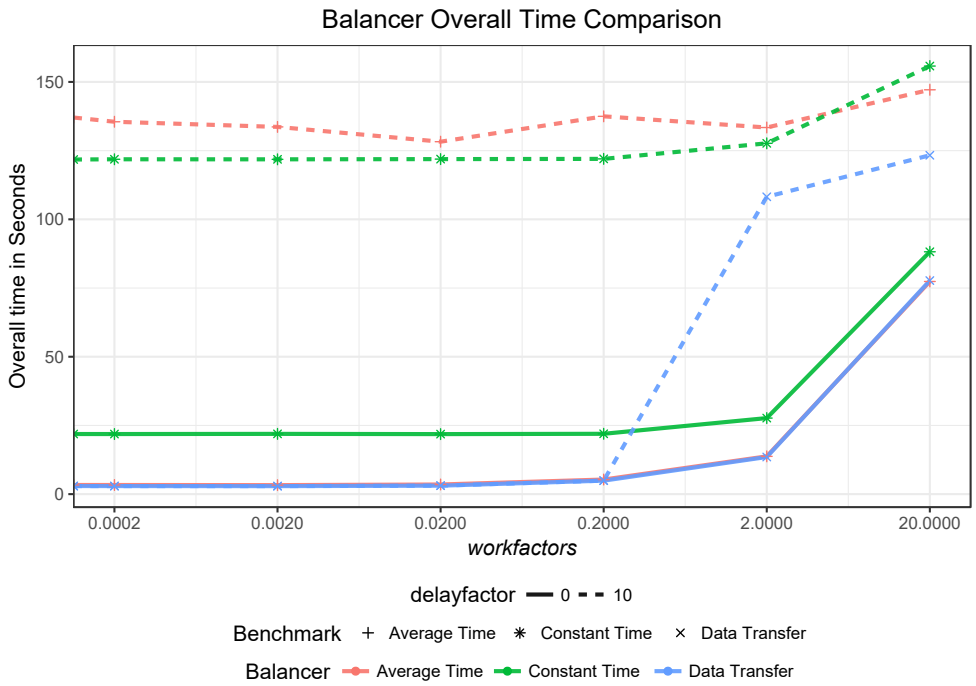


Figure 5.5: Balancer with different *delayfactor*s and *workfactor*s

Conclusion

The Data Transfer Balancer offers a slight improvement in runtime over the Average Time Balancer. This comes at the cost of a more complex interface, an increased need for statistics, and a more elaborate estimation calculation.

5.7 Comparing Transfer Rate

The final benchmark on balancers evaluates the influence different network transfer rates have. The different rates are achieved by implementing a *delayfactor* into the transfer layer that will delay the sending of a message proportional to its size.

The result of this comparison can be found in Figure 5.5. Graphs of the same colour belong to the same balancer, and different line strokes represent different transfer delays.

There are a few things to notice about the results. In the case of no added delay, the performance of the Data Transfer Balancer and Average Time Balancer

are identical, which is a testament to the performance of the used computational cluster and MPI wrapper.

If the *delayfactor* is increased the performance off all balancers degenerates and the Data Transfer Balancer starts to outperform the other balancers. A *delayfactor* of 1 was also measured. However, these results are omitted, as they mirror the shown result and would only clutter the figure.

5.8 Conclusion

As stated at the beginning of this chapter, achieving proper resource utilisation is the primary goal of a balancer. While the use of any of the presented balancers will lead to a correct result, the efficiency of the calculation will vary. Both Data Transfer Balancer and Average Time Balancer have shown promising results in the benchmarks. In fact, if the network transfer is not artificially impeded, they are nearly identical on the used hardware.

6 Plug-ins

The ability of a calculation system to satisfy the needs of its users is limited first and foremost by the operations it provides. After evaluating general performance aspects and ways to distribute arbitrary work over multiple nodes in previous chapters, the focus of this chapter is on the actual calculations and how OPE can be extended with new functionality.

One solution to provide a user with all the required functionality is to offer many elementary operations that can be recombined in any way needed. However, since one of the main goals of OPE is comfortable usability by biologists without programming experience, this approach has its disadvantages. The number of operations needed to be combined to achieve complex operations can be extensive with numerous and confusing connections, essentially requiring the user to program.

Therefore, OPE provides high-level operations. This approach comes at the cost of higher implementation effort for the OPE development. The properties of a desirable operation are:

- A low number of inputs and outputs. This restriction will reduce the complexity for the user and express more clearly what each input/output requires.
- A clear, high-level purpose. Such operations clearly communicate which functionality they provide.

As the number of possible operations is high and specific to the user's application, it is unlikely that a system can provide all of them from the beginning. OPE is thus designed to be extendible, using plug-ins to enhance its capabilities beyond the scope of its basic functions. Going one step further, only core functions are only realised as specialised classes, if they could not be expressed as plug-ins. This approach separates the management aspects from the calculation aspects of the system. The idea of creating small dedicated objects to extend a given system has been a long-standing best practice in software engineering and is described by the open/closed principle [73]. The technology used to realise such plug-ins in OPE is by dynamically loading java jar files at runtime in combination with reflection.

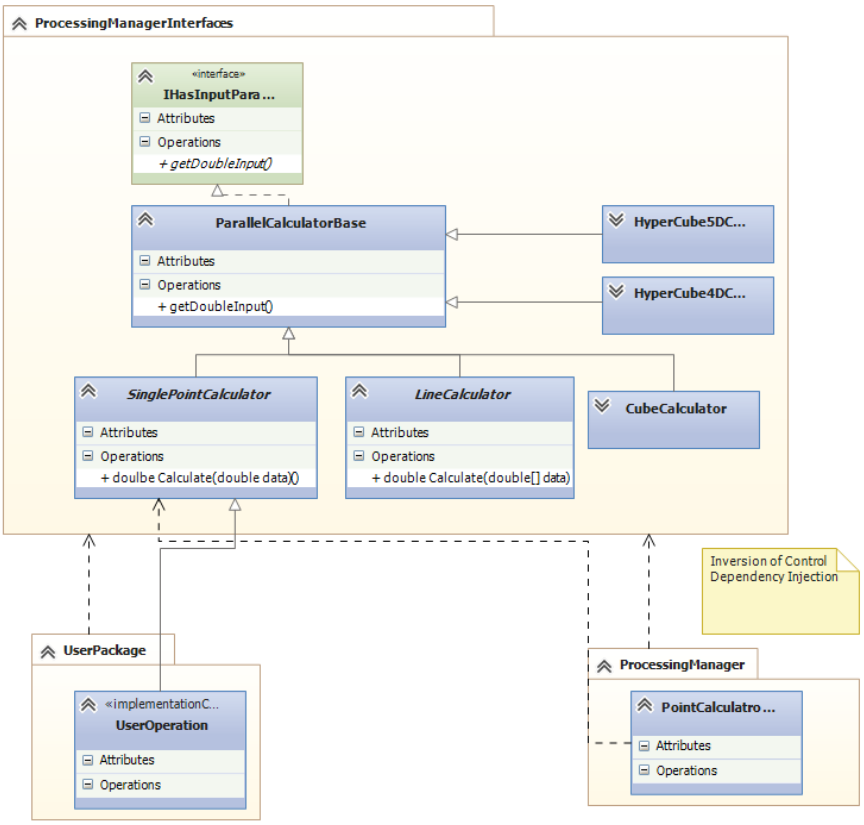


Figure 6.1: Class overview for writing custom plug-ins

6.1 Design Decisions

To achieve the needed flexibility and support plug-ins OPE relies on well-known techniques such as dependency injection and inversion of control [74]. The plug-in interface is provided as a separate jar file, eliminating the dependency of a user plug-in on the processing manager. This design enables the author of new plug-ins to depend only on a compact library that is also very unlikely to change. It also reduces the danger of version changes breaking existing plug-ins. The design is outlined in Figure 6.1. Both user-written packages and the processing manager depend on the `ProcessingManagerInterfaces` package, but not on each other.

Conceptually a plug-in represents the transformation of input data into output

data. As OPE focuses on microscopy images, this shall be first limited to transformations from one image to another. As summarised by Scott Meyers: "Make interfaces easy to use correctly and hard to use incorrectly" [75], the design of the actual plug-in interface follows a number of goals as an orientation:

1) The number of functions to implement by a potential plug-in author should be as low as possible. The author should ideally just be requiring to implement the actual transformation.

2) It should be hard to accidentally break the underlying implementation.

3) Since different images can have different data types, the user should have the possibility to specify the desired data type. If at runtime no version of a plug-in for the actual image data type is available, OPE will choose a version from the available pool and convert the data to the required type. This requirement makes plug-ins more complex, because of the way Java generics are using type erasure [76]. However, standard solutions for this problem exist [77, 78].

   This conversion should also be avoided, as the required type conversion has a non-trivial overhead.

4) The used language features should be as simple as possible. OPE is intended for users with little to no programming experience. Requiring plug-ins to use complicated language features could deter users from writing their own plug-ins.

5) Language features should be used, that allow the compiler to detect faulty plug-in implementations at compile time.

The interfaces designed based on these requirements are realised using an abstract base class. Compared to a pure interface, this implementation allows the inclusion of the code needed for the used The Pointer to Implementation idiom to achieve the already discussed inversion of control.

Each plug-in needs to define which dimensions are required. For example, a calculation on an image plane could require the data along the X- and Y-axis. Alternatively, a vertical cut through a stack could be necessary, requiring data along the X- and Z-axis. This definition is realised by requiring these dimensions as a constructor parameter. While the alternatives of using Java annotations or the implementation of a getter function would have been less verbose, this solution offers the best compile-time checking capabilities.

In addition to the definition of the required dimension, an operation may require additional parameters. These parameters can be specified by using an annotation. Annotations were chosen, as they offer an easy way to not only define the name of the required parameter but also type and default value if desired.

In the following sections, the main plug-in interfaces are explained along with some code examples on how to implement simple functions.

6.2 Plug-ins with constant Dimensions and Sizes

The simplest type of transformation maps one image onto another image of the same size. The base classes available to implement such an operation as a custom plug-in are divided in how many dimensions are required for the calculation:

- `SinglePointCalculator`
- `LineCalculator`
- `PlaneCalculator`
- `CubeCalculator`
- `HyperCube4DCalculator`
- `HyperCube5DCalculator`

The requirements for the implementation are straightforward. A constructor needs to be implemented, which specifies the needed dimension. An implementation of the `Calculate` function is also required. This function contains the actual transformation. All base classes offer no default constructor (without parameters) and define an abstract `Calculate` function. As a result, an incomplete implementation will be detected at compile time. The next sections will show a few simple examples to illustrate how an implementation looks like.

Threshold A common preprocessing operation (see subsection 2.2.2) is to convert an image to a monochrome image based on a threshold value. Ideally, this operation is split into two parts: finding the threshold value and applying it to the image.

In this example, it is assumed that one global known threshold value is supplied for the whole image. This way the example is kept simple and enables the parallelisation

of the calculation in such a way that each pixel can be calculated independently. The code using the `SinglePointCalculator` is given in Listing 6.1.

```
1  // Simple threshold operation
2  // define needed parameter
3  @InputParameter( Name ="Threshold", Typ = Byte.class ,
4          Description = "The value over which the result pixel should be 1 not 0" )
5  public class Threshold extends SinglePointCalculator<Byte>
6  {
7      @Override
8      public Byte Calculate(Byte data)
9      {
10         Byte threshold = getInput ( "Threshold" ) ;
11         return ( data < threshold ) ? ( byte ) 0 : ( byte ) 1;
12     }
13 }
```

Listing 6.1: Threshold

Lines three and four define an additional parameter that is required by the plug-in, as the operation requires the threshold parameter to compute its result. A name, type, and description for this parameter is also provided. The intention is that a description for the block can be automatically generated if it is introduced to the web interface.

The desired base class is selected in line five. As it will work on each pixel individually, this class does not require a derived class to implement a constructor. The implementation of the `Calculate` function starting in line eight is straightforward. The previously defined threshold parameter is acquired and used to decide whether the new pixel value is one or zero. Notice, that the threshold parameter was defined using a Java annotation. While this is not the only solution to tackle this problem it has the benefit of not requiring additional code inside the class, which keeps the implementation tidy.

This implementation already hints at another problem for which no elegant solution was found. The pixel values are set to zero or one, depending on their original value and the threshold. This solution is valid for the `byte` data type. However, if this threshold operation is used in combination with other operations it could be, that following operations assume the values of a thresholded image to be either the minimum or maximum value of the supported colour range.

While it would be easy to solve this problem for this specific operation, it should serve as a hint to a more general problem. More so than when working with plain data, image data suffers from a plethora of formats and implicit conventions inside these formats. Accounting for all possible combination is highly impractical.

Histogram Linearisation As discussed in subsection 2.2.3, histogram linearisation is a technique to improve the contrast of an image by manipulating pixel intensities based on the image histogram. As such, it does require only a single channel of an image, making it a good candidate for parallelisation.

```
// Histogram linearization
public class HistogramLinearization extends PlaneCalculator<Integer>
{
    public HistogramLinearization()
    {
        super(ImageDimension.Y,ImageDimension.X);
    }

    @Override
    public Integer[][] Calculate(Integer[][] data)
    {
        Set<Integer> foundIntensities = new HashSet<>();

        Integer maxIntensity = (int)this.getMaxValue();
        Integer minIntensity = (int)this.getMinValue();
        for (Integer[] line : data)
        {
            for (Integer point : line)
            {
                maxIntensity = Math.max(maxIntensity,point);
                minIntensity = Math.min(minIntensity,point);
            }
        }
        double dataRange = (double)maxIntensity- (double)minIntensity;
        double intRange = (double)Integer.MAX_VALUE - (double)Integer.MIN_VALUE;

        // rescale Image
        Integer[][] result = TemplateHelper.CreateNewWithSameSize(data);
        for (int i=0;i<data.length;i++)
        {
            for (int j=0;j<data[0].length;j++)
            {
                int current = data [i][j];
                result[i][j] = (int) (((current - minIntensity) / dataRange) * intRange);
            }
        }

        return result;
    }
}
```

Listing 6.2: Histogram Linearisation

For this example, it is assumed that the linearisation should be applied for each XY-plane separately. An implementation using `PlaneCalculator` base class is given in Listing 6.2.

There are two major differences when compared to the threshold example. First, as the linearisation does not require a parameter, so none is specified. Second, `HistogramLinearization` implements a constructor starting in line four. As `PlaneCalculator` requires a definition of what plane to calculate, the base class constructor requires two dimension parameters.

As with the Threshold operation, this implementation shows the problems of having to support multiple image types. As the presented implementation is for an `Integer` data type, a naïve implementation would rescale the colour spectrum to the range of (`Integer.MIN_VALUE`, `Integer.MAX_VALUE`). If no other implementation is provided and a `Double`-typed image being processed by this implementation would therefore be rescaled to the wrong minimum and maximum values, as OPE tries to convert data types automatically. As a solution, the minimum and maximum intensity values can be queried as seen in lines 14 and 15. However, each additional helper function introduced to solve a given format problem makes it more difficult for a potential user to write clean code.

6.3 Projection Plug-ins

The other big type of plug-ins are such, that project data along a given image dimension. The resulting image, in turn, has a reduced size of one in all the projection dimensions and keeps its size in all other dimensions. These plug-ins have to implement only the reduction function [79]. As an example operation, Listing 6.3 shows how a MIP (see subsection 2.2.4) is implemented in OPE. The base classes follow a naming schema which calls the reduction a projection, as the only relevant operation found and implemented was the MIP. As such, reduction and projection are considered interchangeable for the rest of the work.

In addition to specifying the dimensions needed to calculate the results, as introduced for plug-ins with constant dimensions and sizes, the base classes for projection plug-ins also require a definition of the dimensions in which the projection will take place. As before, this definition is implemented by calling an abstract base class constructor. This constructor expects two lists: one for the dependency dimensions and one for the projection dimensions. Where possible, this constructor has been simplified. Given the example of a MIP along the Z-axis: Since it is a point-based operation, it has no dependency dimensions. As such the used constructor requires only one dimension, the projection dimension. There is currently no operation available to project multiple dimensions at once. If needed, it can be achieved

by executing the individual projections after another. Alternatively, OPE can be extended to support such an operation.

```
// Reduce a stack to a single slice containing the maximum intensity for that point/channel
@SuppressWarnings("unused") // class will be found through reflection
public class MaximumIntensityProjectionZ extends PointProjector<Double>
{
    public MaximumIntensityProjectionZ()
    {
        super(ImageDimension.Z);
    }

    @Override
    public Double Aggregate(Double data1, Double data2)
    {
        return Math.max(data1,data2);
    }
}
```

Listing 6.3: Maximum Intensity Projection

One possible underlying implementation for the projection is to split the image into slices along the Z-axis. The resulting Z-stacks, with a one-pixel area, are then stepwise reduced using the provided `Aggregate` method. The advantage of this approach is that the operation can be executed in one calculation step for each Z-stack.

For more complex operations, this solution becomes problematic. As an example, the tracking of a particle across multiple time points could be expressed as a projection of full XY-planes along the T-axis. For a single channel image, with no Z information, such an allocation would imply the need for the complete image. As a result, no parallelisation can be applied, even though the operation is expressed in a way that would support parallelisation.

Therefore, OPE uses this data splitting method only for simple operations, such as point projections. For more complex operations, partial stacks are reduced in parallel, with their result forming new stacks, which can be split again. This splitting and reducing is repeated until only a single plane remains. As a result, the execution follows a more efficient tree pattern. However, this implementation incurs additional overhead, as intermediate results are being generated. Furthermore, the execution logic of OPE has to be more elaborate, as the projection operation needs to re-ingest their outputs containing partial results until the operation is finished.

As another example of an operation used in chapter 7, Listing 6.4 shows the implementation of the time point selection operation. The base class used is `IndexedPlaneProjector` which requires full planes. Consequently, the constructor starting

in line five specifies that the projection needs a full XY-plane. Line eight additionally states that the projection will happen along the T-axis. The requirement for full XY-plane in this implementation is not because of algorithmic requirements, but rather to reduce the number of copy operations.

Listing 6.4 also shows an additional performance-related implementation detail. To avoid automatic type conversion, this class provides multiple `Aggregate` functions for different types. The selection of the correct implementation is handled by OPE. Implementing multiple versions should be seen as an optional possibility to improve performance. However, doing so is contradictory to the goal of a simple, easy-to-understand implementation. As with the Threshold and Histogram Linearisation operations, the need to offer these facilities arises from the multitude of possible image formats. On a final note, the used `IndexedPlaneProjector` base class passes two additional parameters to the `Aggregate` function. These parameters describe the origin of the data, as the implemented function needs to make decisions based on this origin. While this particular additional parameter might be used rarely, it hints at the need of may operations to require additional information.

```
// Extract a single time point from a time series
@InputParameter(Name = "TimeIndex" , Description = "The time point to select" , Typ =
      int.class)
public class TimePointSelectionPlane<T> extends IndexedPlaneProjector<T>
{
    public TimePointSelectionPlane()
    {
        super(ImageDimension.X,ImageDimension.Y, // 2x dependency, as it extends
              plane-projector
                ImageDimension.T);    // 1x projection, as only one is currently supported
    }

    @Override
    public T[][] Aggregate(T[][] data1, T[][] data2, int posTData1, int posTData2)
    {
        int desiredT = this.getInput("TimeIndex");
        T[][] result = TemplateHelper.CreateNewWithSameSize(data1);
        T[][] source = data1;
        if (posTData2== desiredT)
        {
            source= data2;
        }

        for (int y = 0; y < data1.length; y++)
        {
            System.arraycopy(source[y],0,result[y],0,source[y].length);
        }
        return result;
    }

    public byte[][] Aggregate(byte[][] data1, byte[][] data2, int posTData1, int posTData2)
    {
       ...
    }

    public float[][] Aggregate(float[][] data1, float[][] data2, int posTData1, int
          posTData2)
    {
       ...
    }
}
```

Listing 6.4: Time Point Selection implementation with projection

6.4 External Tool Plug-ins

As many workflows use already existing tools, it is important to consider how these tools are handled. Reimplementing them for OPE seems practical only for the simplest of tasks. Beyond that, a mechanism to utilise external tools is provided. The

most basic support comes from the `ExternalToolBlockBase` class, which is used to build plug-ins that transform one image into another without changing the size using an external tool. An example of a block build upon it is the `CopyExternalToolBlock` presented in Listing 6.5. This example does not execute any calculation and only forwards the input to the output.

```
1  // Do nothing but copying the data.
2  public class CopyExternalToolBlock extends ExternalToolBlockBase
3  {
4      @Override
5      public void RunCalculation(String pathToInputFile, String pathToOutput)
6      {
7          String command;
8          if (System.getProperty("os.name").startsWith("Windows"))
9          {
10             command= "cmd.exe /c copy " ;
11         }
12         else
13         {
14             command = "cp";
15         }
16         this.CallProgram(command + " " +pathToInputFile + " " + pathToOutput);
17     }
18 }
```

Listing 6.5: Copy Data via external copy operation

All external tool plug-ins work by exporting the input image to a file. A path to this temporary file is then provided to the plug-in implementer, as well as a path to where the generated output file must reside after executing the external tool. After the external tool was called by the plug-in, this result file is automatically re-imported as an image into OPE. In the case of the example, the only thing to consider is the different names of copy commands on different platforms.

While the possibility to use any external tool increases the flexibility of OPE, it reintroduces a problem that was already discussed in the introduction. As the command call is delegated to the operation system, OPE can no longer track or manage the actually used tool and its version. To remedy this shortcoming, plug-ins that are expected to be based on tools that will change versions can specify where the needed version of the tool can be found. The intended storage is, as with versioned plug-ins, a version control system such as Git. Listing 6.6 shows an example of this solution, in combination with the DeconvolutionLab2 tool and the `ExternalToolBlockBase` class. As with the versioned plug-ins, it is the responsibility of the web interface to download the correct version and incorporate it into the deployment environment (see section 4.2).

An alternative to this approach of committing the tool into a version control

repository is the use of virtual machines or similar technologies. Especially the use of Docker has recently become popular [80, 81, 82], going as far as creating separate repositories for bioinformatic applications [83]. The advantage of this approach is the small size of Docker container descriptions, as they include only the steps needed to install all necessary software packages. In contrast, committing the tools binary to a version control system incurs a significantly higher need for data storage. On the other hand, the time overhead needed to instantiate such a container makes them ill-suited for the representation of small tools. Consequently, Docker is a reasonable solution for representing complex operations or workflows and ensuring their reproducibility. However, the time overhead makes Docker a suboptimal choice for OPE. Nevertheless, in certain use cases, using containers can be the only practical way to include a tool, especially if a tool has many dependencies, that cannot be adequately be represented in a single archive file. The general rule is that tools which are packaged as jar files are easy to handle. Python-based tools tend to be more complicated in their installation as they often depend on a number of other packages.

On a final note, using Docker introduces the need for a network connection. As with accessing version control, this would require the web interface to download the required Docker containers and deploy them to the cluster.

As with most of the discussed algorithms, many tools provide functionality that can be parallelised. Some of these tools offer the option to do so, others do not. And while the uses of multiple CPUs is not uncommon, using multiple compute nodes is the exception. Additionally, the rare cases that provide multiple node support require additional configuration to work correctly.

By using these tools as a plug-in in OPE and specifying the inherent, a priori known parallelism even tools that do not support parallelism out of the box can be automatically parallelised. Listing 6.7 shows the modifications that have to be made based on Listing 6.6 to achieve such a parallel plug-in. By exchanging the base class with `PlaneExternalTool` and specifying that each XY-plane can be calculated independently in the constructor, a parallel version of the deconvolution tool used in this example is created.

6.5 External Non-Image Plug-ins

The plug-ins discussed up till now transform one image into another. This subset of problems is sufficient to implement the pre-processing workflows outlined when

```
// non parallised deconvolution using DeconvolutionLab2
public class NonParallelDeconvolutionTool extends ExternalToolBlockBase
{
    private final static String JarName = "DeconvolutionLab2custom.jar";

    static final SourceControlInfo info =
           new SourceControlInfo("GIT",
                  "https://git.inf-ra.uni-jena.de/xo46rud/OpeToolBinaries/raw/"+
                  "master/DeconvolutionLab2/" + JarName,
                  "f4d64df8b3a8af8fc40e6b59b2888114fcc51f3d");

    NonParallelDeconvolutionTool()
    {
        super(info);
    }

    @Override
    public void RunCalculation(String pathToInputFile, String pathToOutput)
    {
        String command = "";
        command+= "Run -image file "+ pathToInputFile;
        command+= " -psf synthetic Double-Helix 3.0 30.0 10.0 size 30 30 10 intensity 255.0";
        command+= " -algorithm RIF 0.1000";
        command+= " -out mip " + pathToOutput;
        this.CallJar(JarName, command ); // use helper function from base class
    }
}
```

Listing 6.6: Deconvolution via external Tool

considering the initial design of OPE (see section 4.1). To explore a more complex application in chapter 7, it is also necessary to handle non-image results. The detection and measurement of cells (see section 2.3) within a time series is a popular operation and is used to illustrate this problem. The result of this operation is a table containing the positions and statistics of the cells for each time point. The challenge with such an operation is to find a solution to generalise the result in such a way, that other work blocks can use the result. As the detection of the cells is a long-standing problem a range of well-accepted tools, such as CellProfiler, exist. Using them simplifies the problem, as most offer the option to save the result data to a Comma-Separated Value (CSV)-file, which can be easily imported and exported. OPE provides a representation for tables, which accepts such a CSV-file. This representation also hides the distributed storage of such a table behind an abstraction.

To illustrate how such a tool is added as a plug-in, Listing 6.8 shows a possible implementation. There are a few things to notice here. First, different tools require different ways in which the input and output data is provided. These variations require a more granular configuration of plug-ins. In the case of CellProfiler, the data is required to be passed as a folder, not a single file. Hence the additional parameters

```
1  // non parallised deconvolution using DeconvolutionLab2
2  public class ParallelPlaneDeconvolutionTool extends PlaneExternalTool
3  {
4      private final static String JarName = "DeconvolutionLab2custom.jar";
5
6      static final SourceControlInfo info =
7              new SourceControlInfo("GIT",
8                  "https://git.university-git.org/User/OpeToolBinaries/raw/"+
9                  "master/DeconvolutionLab2/" + JarName,
10                 "f4d64df8b3a8af8fc40e6b59b2888114fcc51f3d");
11
12     ParallelPlaneDeconvolutionTool()
13     {
14         super(info, ImageDimension.Y,ImageDimension.X);
15     }
16
17     @Override
18     public void RunCalculation(String pathToInputFile, String pathToOutput)
19     {
20         String command = "";
21         command+= "Run -image file "+ pathToInputFile;
22         command+= " -psf synthetic Double-Helix 3.0 30.0 10.0 size 30 30 10 intensity 255.0";
23         command+= " -algorithm RIF 0.1000";
24         command+= " -out mip " + pathToOutput;
25         this.CallJar(JarName, command );
26     }
27 }
```

Listing 6.7: Parallel Deconvolution via external tool

in the constructor are necessary. Second, in contrast to working with images, it is not always clear how different table parts should be merged. As the external tool is presented with a sub-image without any context, it cannot correctly fill the necessary entries in the table. As an example, CellProfiler records the data origin in a column it calls “ImageNumber”. Since each worker will present CellProfiler with only a single sub-image, this column will always have the value one. The example implementation solves this problem by instructing OPE to create an additional column to track the data origin. This directive is part of the `ToolInputOutputConfiguration` in line 8. This is done for the Z- and T-coordinate, since CellProfiler requires the other dimensions to run its calculations.

6.6 Conclusion

This chapter illustrates the different ways to design and implement extensions in the form of plug-ins for OPE. As the operations are getting more and more complex, the amount of additional information that has to be expressed continuously grows. Information on which dimensions are needed to execute the calculation is only sufficient for basic operations. For more sophisticated calculation, additional information

```
// Cell statistics using the cellprofiler tool
public class CellProfilerTool extends CubeExternalTool
{
    public CellProfilerTool()
    {
        super(null,
               ToolInputOutputConfiguration.UseFileFromOutputFolder("Cells.csv",
                    ToolInputOutputFileType.CSV).UseInputFolder(),
               TableMergePostprocessing.CreatePositionColumn(),
               ImageDimension.Y,ImageDimension.X,ImageDimension.C);
    }

    @Override
    public void RunCalculation(File pathToInput, File pathToOutput)
    {
        String command;
        if (System.getProperty("os.name").startsWith("Windows"))
        {
            // windows specific code for development
        }
        else
        {
            command = "cellprofiler -c -r -i " + pathToInput.getAbsolutePath() + " -o " +
                 pathToOutput.getAbsolutePath() + " -p
                 /path/to/pipeline/ExampleHumanForComposit.cppipe";
        }

        this.CallProgram(command);
    }
}
```

Listing 6.8: The use of Cellprofiler to measure cells

about the image format and the data origin is needed.

As was implicitly assumed when initially restricting the focus of OPE to images, the complexity of the plug-ins drastically increases when non-image types are introduced. As shown in section 6.5, even the handling of well-standardised CSV-files requires considerable more information and implementation effort. This problem is aggravated further if less structured data formats have to be considered.

Another big issue is the support of external tools. Fundamentally, storing the correct version in an archive and retrieving it when needed should ensure that the usage is reproducible. However, depending on the used technology and the decisions made in its design, external tools can require a complicated set of dependencies and a particular way in how to interface with them. As a result, it can be necessary to extend the plug-in facilities of OPE with each new supported tool. Approaches like using virtualisation can be used to remedy these problems, but they incur additional overhead and tend to require elevated access rights.

Still, OPE's plug-in infrastructure ensures reproducible and parallel calculations. The only area where concessions had to be made is the integration of external tools.

7 Application

The previous chapters outline the design of OPE, evaluating how it performs in its intended deployment environment and illustrating how it can be extended with additional functionality. What is missing is for all these parts to come together in an application scenario. For this demonstration, a workflow is used which closely resembles what an actual experiment workflow of a biologist may look like. While this workflow omits some of the more detailed steps to make it easier to comprehend, it consists of commonly used tools and operations.

7.1 Example Workflow

The starting point of the workflow is an experiment in which a camera combined with a microscope is used to capture the development of cells over a time span. A number of processing steps is executed following this image acquisition. These steps analyse the data as well as preparing it for a potential publication. Therefore, the workflow is roughly separated into two parts: data analysis and improving the image quality. A graphical representation of the workflow is depicted in Figure 7.1.

The upper half of Figure 7.1 shows the data analysis part. It starts with improving the data with deconvolution, followed by measuring cells and generating a plot based on these measurements. The lower half of the figure shows the image quality improvement operations. Here, single time point is selected and beautified by removing noise and transforming the colours. All steps will be explained in more detail in the following sections, staring at the input and following the flow of data.

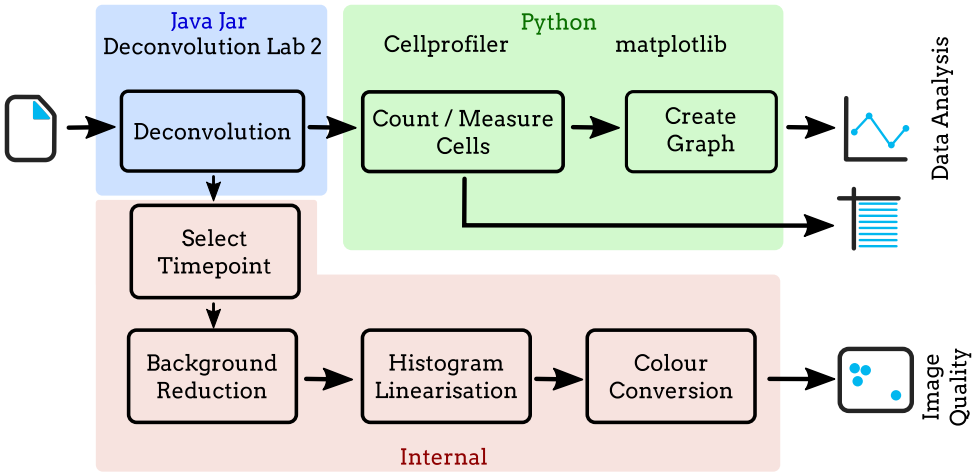


Figure 7.1: Example application workflow, icons from and based on [84]

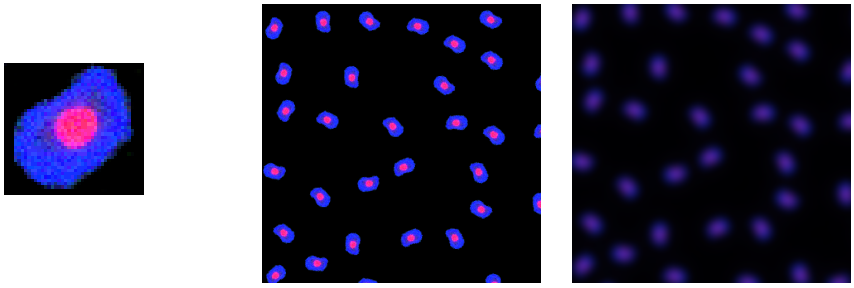
7.1.1 Source Data

The example idea is borrowed from the field of cell biology. It is assumed that a cell culture was observed over a time span using a microscope with a relatively small magnification. This way as many cells as possible can be observed at once. In combination with modern high-resolution cameras, these cells are still large enough to make out a sufficient amount of detail for statistics on the cell shape.

The actual data used in the measurements were artificially constructed to be able to check the actual results against the calculated ones. The images were created by extracting the image of a single cell from the CellProfiler "Human cells" example. This single cell was duplicated and pasted onto random positions of a larger image, without overlap and with random rotation. Example images illustrating these steps are shown in Figure 7.2.

Uniformly distributed random noise in the interval $[0, 5]$ was also added to the image. This noise simulates noise a detector would introduce. The upper bound of five is later used in the workflow as a parameter for the background reduction operation.

As the used example cell was from a confocal image and as such very sharp the whole image was then convolved using a generated PSF to accommodate for the effects a microscope would have on a wide-field image. This PSF was generated using DeconvolutionLab2's PSF-Generator using the Born&Wolf model for a refractive index of 1.5, a wavelength of 610 nm, and pixel size of 10 nm [85].



(a) Single cell extracted from CellProfiler example "Human cells"

(b) Sub-image of the constructed input

(c) Convolved version of sub-image

Figure 7.2: Steps in constructing the input data

Notice, that the convolved version of the image (Figure 7.2c) is considerably darker compared to its origin data. The reason being, that the same intensity information is spread out over a larger area than in the original image. The argument can be made that a biologist would thus increase the brightness during image acquisition. This step is omitted here, as it could interfere with the deconvolution step and is also compensated for by using a histogram linearisation operation in the workflow.

Finally, a time-series image was created by selecting and concatenating multiple convolved images. This way the number of images used can be varied by adjusting the time series length and input data size.

7.1.2 Data Analysis

The data analysis side of the workflow focuses on the question of how the number of the cells changed over the course of the experiment. The desired result is a graph. The first step in this process is to run the complete data through a deconvolution tool (see subsection 2.2.1). The experimenter in this example will use a relatively small magnification to track as many cells as possible. This low magnification in combination with the high-resolution of modern cameras provides an adequate overview image while still keeping the individual cells big enough to collect meaningful statistics on them.

However, before this analysis can be done, deconvolution should be used to remove the effect of the microscopes PSF on the data. As this is a reasonably common problem, OPE will use the popular DeconvolutionLab2 software package to execute

this step. The PSF data required for this step is the same as was used to blur the image in the first step.

After sharpening the image with deconvolution, the cells in the image are counted. As before, this is a long-standing problem and a range of tools exists. In this example, the popular CellProfiler tool is used to measure the individual cells. The processing pipeline used for this measurement is based on the pipeline provided in the CellProfiler "Human cells" example, which was also the source of the cell images. This way the pipeline matches the cells seen in the data.

As a final step in evaluating the data, a graph is generated using a Python script and the matplotlib library. As more information, such as cell size and different shape indices are included in the data, more complex analysis scripts would be possible. However, for simplicity reasons, the used script will only plot the cell count.

Results The final cell count measurement of a workflow execution is shown in Figure 7.3, together with the expected graph resulting from the image generation step. The cell count function to generate the input images is, in essence, arbitrary as it merely needs to show that the expected result matches the actual result. Because of this freedom, it was selected to "look nice" by selecting a few points in an X-Y-plane and fitting a quartic equation. The resulting equation is rather cumbersome and only mentioned here for the sake of completeness. The significant point to make is that the expected data points and the measured data points line up.

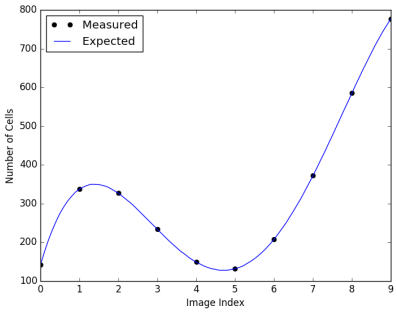
The equation used for the data is:

$$f(x) = 1000\frac{1 + 0.0054 - 6.2 \times 10^{-6}x^2 + 2.2 \times 10^{-9}x^3 - 2.3 \times 10^{-13}x^4}{7}$$

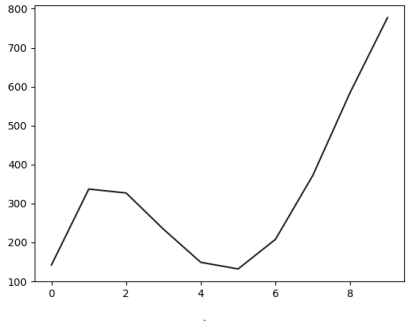
refitted from the interval $[0, 4500)$ to the interval $[0, 9)$. The interval of $[0, 9)$ is chosen, as the used time-series was constructed with ten time points

7.1.3 Improve Image Quality

The goal of improving the image quality is to create an image which will represent the experiment. This image does not necessarily have to be suitable for quantitative analysis. As such, operations which rebalance or modify colour or content are applicable. The user will have to pick a time point to use for this half of the workflow, since processing the complete image is unnecessary. The improvement steps will be based on the result of the deconvolution, as that step also represents



(a) Expected cell count graph including measured cell counts



(b) Graph image generated during execution in OPE

Figure 7.3: Expected and measured cell count graphs

an improvement in image quality.

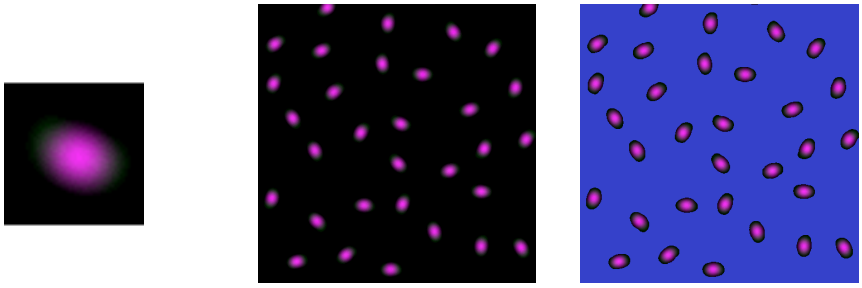
First, the background containing only noise is removed. This is achieved by reducing the intensity of all pixels in the image by some offset value (to a minimum of zero). If the offset is well selected, this transformation will lead to all background pixels beeing set to zero, while keeping the cells intact.

Second, since the shift operation narrows the used histogram the next operation is a histogram linearisation (see subsection 2.2.3). The resulting image fully utilises the intensity range provided by the image format, which results in better contrast and human readability of the image.

After these steps, other standard steps could follow. For example, sharpening the image or adding a scale bar. However, to keep the example at a manageable size, these step are omitted.

The final step of the image improvement is to shift the images colour pallet from the RGB to a magenta green based pallet. This step is sometimes required in publications that address potentially red-green colour blind readers.

Results A small sub-image of the whole improvement process is shown in Figure 7.4. The increased brightness is a result of the deconvolution and the histogram linearisation. The shape of the cell nucleus was not reconstructed as sharp as it was in the original image. However, this blurring is to be expected and good enough to get rough measurements. The colour of the cell itself has changed from blue to green because of the colour conversion. However, this change is hard to see in the



(a) Single cell in the result image

(b) Sub-image of reconstructed and improved input

(c) Background changed to blue to visualise noise removal

Figure 7.4: Results of the image improvement process

composite image as the cell nucleus occupies a proportionally large area. Finally, emphasising all pixels that are zero with blue in Figure 7.4c shows that background noise was removed from the image.

7.2 OPE Workflow

After outlining the used steps, the workflow is created in OPE using the web interface. The graphical representation of the resulting workflow is shown in Figure 7.5.

All processing blocks are straightforward and mirror the operations already described. Of notice are the input and output blocks, as they define the interactions with OMERO. The image enters OPE through a "Load Image" block, that is provided with the OMERO ID of the image by the user. The web interface supports

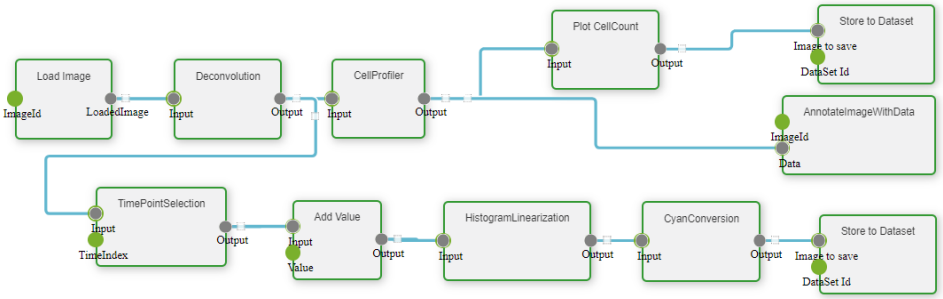


Figure 7.5: Example application workflow in web interface

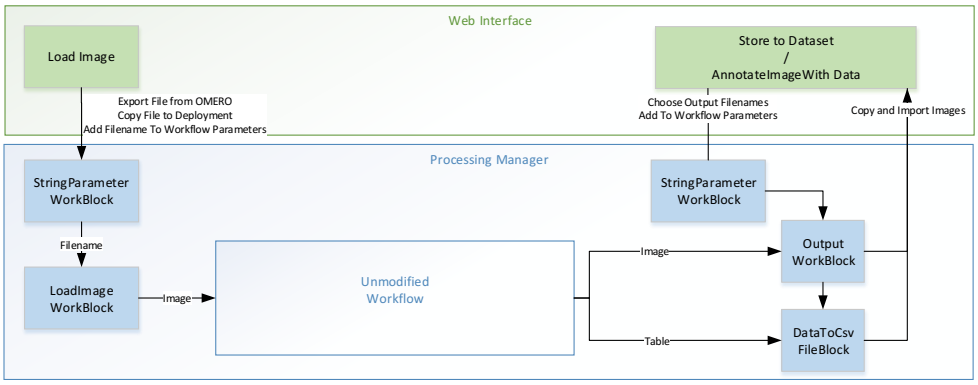


Figure 7.6: Graph transformation as executed by processing manager

the user in this selection by providing a convenient graphical tool.

As for the output, two different output blocks are used: "Annotate Image With Data" and "Store to Dataset". The distinction is necessary as OMERO datasets can only hold images and the output of the "CellProfiler" block is a table stored in a CSV file. Such non-image files have to be annotated onto either an image, dataset, or project to be easily accessible via an unmodified OMERO.

Before executing this workflow, it is modified by OPE. The computing environment used by the processing manager is restricted in its network access and can thus not directly access OMERO. To circumvent this limitation, the web interface will export the image from OMERO to a file and re-import the result files, enabling the processing manager to work with files as its inputs and outputs. To facilitate these changes, the processing manager will replace all input blocks with a "StringParameterWorkBlock" holding the input file name and a "LoadImageWorkBlock". The output blocks are replaced, again, with "StringParameterWorkBlock"s holding an output file name each, and an "OutputWorkBlock" for images and a "DataToCsvFileBlock" for tables. A schema describing these interactions and transformations is shown in Figure 7.6.

7.3 Parallelisation

One of the goals of OPE is to parallelise a given workflow. To understand what that could mean for the example workflow a short analysis of how the separate operations can be performed is given in Table 7.1. Three general types of parallelism can be

Plug-in Operation	Parallelisation	Tool Parallelism	Plug-in Type
Deconvolution	X-Y Planes	Multi Core	ParallelExternal
Count/Measuring Cells	X-Y-C Planes	Multi Core	ParallelExternal
Create Graph	needs full table	none	External
Pick Time Point	Plane Projection	OPE internal	Reduction
Background Reduction	Pixel Independent	OPE internal	PointCalculator
Histogram Linearisation	X-Y Planes	OPE internal	PlaneCalculator
Colour Conversion	Pixel Independent	OPE internal	PlaneCalculator

Table 7.1: Parallelism supported by the example workflow steps

found.

1) The creation of the graph is a single threaded Python application and requires the complete input data at once.

2) Both external tools are optimised to run a single image utilising as much of the computing power the local machine has to offer.

3) The internal tools are designed to be distributed over multiple worker nodes, making the actually used parallelism dependent on factors such as the used balancer, computational complexity, and network speed.

One somewhat counter-intuitive realisation comes from the fact that the extraction of a single time point is implemented as a reduction. This is not the most efficient way of implementing it, as reductions can be executed as sub-chunks, requiring multiple passes even though the selection requires no actual calculation in most chunks. As an additional effect, this way of implementation can lead to unnecessary duplication and movement of image data.

Two factors led to that decision. The desire to express even this block inside the constraints provided by the plug-ins and the fact that a non-parallelised implementation, while faster, would require the complete image in memory. Especially the memory requirement is non-trivial considering the size of the potential images, which is also amplified by preceding deconvolution step which increases the bit depth of the image.

Another point to make is that neither CellProfiler nor DeconvolutionLab2 offer native support for parallelisation across multiple nodes. However, the data dependencies of the algorithms involved are known and the plug-ins used to access these tools include this information. With this knowledge, OPE can automatically split

the images and process different parts on different nodes, finally merging the results back together. This approach allows the use of optimised tools while fully utilising the capabilities of the execution environment.

7.4 Runtime Model

Before examining actual time measurements, a short analysis of the expected results is given. The overall time of the execution ($T_{overall}$) is composed of the time needed to load the image, deconvolution, data analysis (T_A), and image quality improvements (T_I), as well as storing the results. Given enough computing nodes, data analysis and image quality improvements, as well as their respective storage operations, can be run in parallel.

$$\begin{aligned} T_A &= T_{Cellcount} + T_{DrawGraph} \\ T_I &= T_{TimeSelection} + T_{Add} + T_{Linearisation} + T_{ColourConv} \\ T_{overall} &= T_{Load} + T_{Deconv} + T_A + T_I + T_{Save} \end{aligned}$$

As the parallelisation for the external tools is known, it can be expressed in respect to the number of compute nodes n used in the calculation as well as the number of time points t to be processed in the input image. The difference between both tools is that the deconvolution happens channel wise, whereas the cell counting requires complete time points. It is assumed that the input data is an RGB image and has three channels.

$$\begin{aligned} T_{Deconv}(n) &\approx \left\lceil \frac{3t}{n} \right\rceil T_{DeconvSinglePlane} \\ T_{Cellcount}(n) &\approx \left\lceil \frac{t}{n} \right\rceil T_{CellcountSingleT} \end{aligned}$$

Another factor influencing the actual parallel behaviour is the order in which the workflow describes the work blocks, as no new block is started as long as the blocks currently being processed saturate the available workers. To not overcomplicate the analysis, it can be assumed that the workflow was set up in a way that the execution of the image analysis will preempt the execution of the image improvement.

With this, a rough analysis of the time and parallelism behaviour follows the

following steps:

1) Loading the input data cannot be parallelised and is handled by a single node. It thus incurs a constant overhead.

2) Deconvolution can be parallelised on a per channel basis, profiting from up to $3t$ compute nodes.

3) Profiling of the cells can be parallelised on a per time point basis, profiting from up to t compute nodes. If more nodes are available the next ready operation (time point selection) will fill the remaining nodes at the discretion of the used load balancer.

4) Plotting the cell count will use a single node, as will outputting the cells statistic to CSV-File.

5) Saving the final graph is done by a single node. This constitutes the end of the analysis side of the workflow and the image improvement side will start.

6) All internal operations used in image improvement will fill available nodes alongside steps 3 to 5, as seen fit by the load balancer.

7) One node is used as a master node and will not participate in the calculation.

With the exception of the time point selection, all operations in the image improvement side of the workflow are computationally much simpler than the image analysis side. It thus seems reasonable to assume, that they can be neglected when considering the overall runtime. The storage of the cell statistics to a CSV-File in step 4 can also be omitted as the simultaneously running plotting tool will also include a storage to file operation, to pass the table to the external Python script. To a lesser degree, in general, the output of the results can be omitted, as the resulting images (a graph and a single slice) are considerably smaller than the data used inside the workflow. They also occur at the end of both workflow sides and can be treated as another part of a constant overhead.

As such, a rough estimation of the overall time can formulate as:

$$T_{overall}(n) \approx T_{ConstOverhead} + \left\lceil \frac{3t}{n} \right\rceil T_{DeconvSinglePlane} + \left\lceil \frac{t}{n} \right\rceil T_{CellcountSingleT} \tag{7.1}$$

Finally, the benchmark only varies the number of nodes, using the same image for each measurement. This means that the number of time points t is constant.

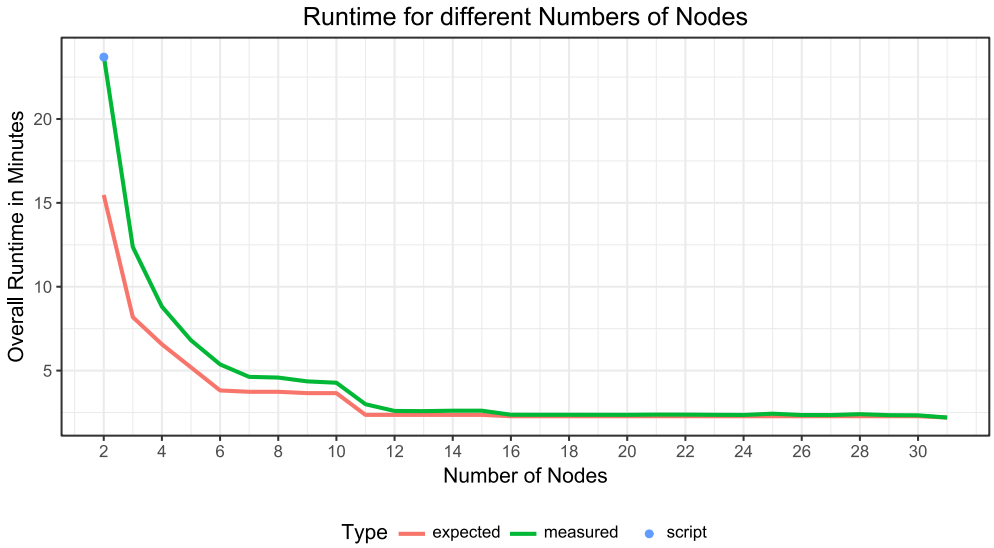


Figure 7.7: Experimental results of application workflow vs expected runtimes

Both major components can be and have been measured individually without the influence of OPE, the result of this measurement are listed in Table 7.2.

7.5 Experimental Results

The experimental results were measured using an image with ten time points, utilising 2 to 32 nodes on the Ara Cluster. Each measurement was repeated at least 100 times and the average time was calculated. The results are shown in Figure 7.7. The time needed for running CellProfiler and DeconvolutionLab2 on one of their work chunks has also been measured by executing the respective command outside of OPE 100 times and calculating the average of the execution time. These results are listed in Table 7.2. The resulting measurement graph consists of the following phases:

- As the profiling of the cells takes very long the improvements up to 11 nodes are very rapidly. At this point (11 nodes means 10 worker nodes) all time points are analysed in parallel by CellProfiler. Since DeconvolutionLab2 has to calculate three times the number of work chunks, it is still not fully parallelised.
- Starting at 12 nodes, the image improvement side of the workflow can start in

Work block	Individual Time measured
CellProfiler	76.5 seconds
DeconvolutionLab2	4.8 seconds

Table 7.2: List of individual work block time measurements

parallel to the cell profiling.

- Up to 31 nodes more and more data slices are deconvolved at once. However, the improvements in runtime with each additional node are not as significant, as 11 nodes are enough to process the image in three rounds, 16 for two rounds and 31 for one round.

Evaluating Equation 7.1 and setting $t = 10$ as well as using the measured times from Table 7.2 the expected curve follows:

$$T_{overall}(n) \approx T_{ConstOverhead} + \left\lceil \frac{30}{n} \right\rceil 4.8\,\mathrm{s} + \left\lceil \frac{10}{n} \right\rceil 76.5\,\mathrm{s} \tag{7.2}$$

A simple way to estimate the overhead is to fit the theoretical curve to either the beginning or end of the measured curve. Fitting to the beginning of the curve has the disadvantage of violating some of the simplifying assumptions as to what can be executed in parallel. Because of this problem, the overhead used in the reference function has been fitted to match the tail of the measured values. In this case, the calculated overhead is 50 s.

Both the expected and measured function match up rather nicely. As expected, the most significant deviation occurs with low numbers of worker nodes. The reasons can be found in simplifications of the expectation model such as time point selection, image improvement, and storage of the cell statistic CSV being handled in parallel for a sufficient number of worker nodes. Also, the times in Table 7.2 have been measured while always using the same input chunk. In the complete workflow, depending on the distribution and order of the chunks some additional runtime fluctuations can occur, as both CellProfiler and DeconvolutionLab2 are dependent on the image content.

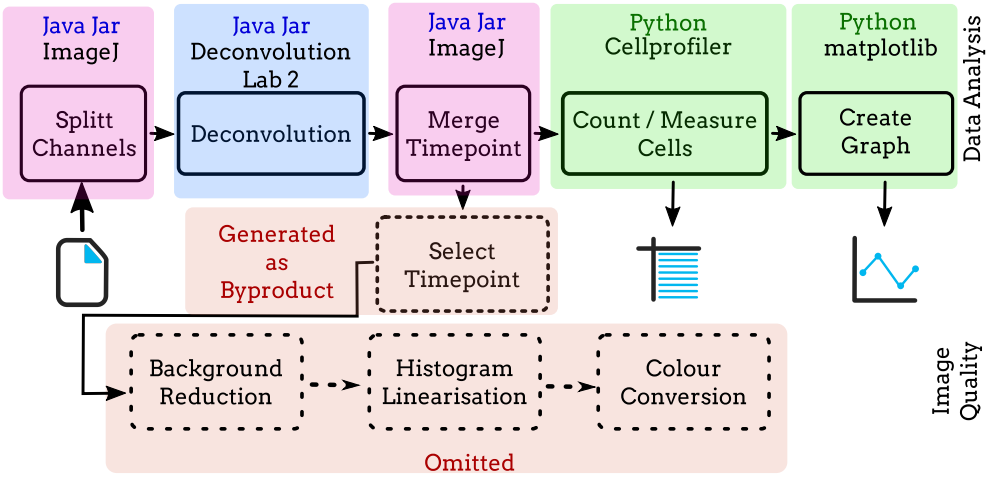


Figure 7.8: Command line workflow equivalent to application workflow

7.6 Script Comparison

In an effort to better understand and assess the results of the previous section, the workflow was recreated using a shell script and standard image processing tools. The overview can be found in Figure 7.8 and the measured time is also shown in Figure 7.7. The steps executed by the script are as follows:

1) Split the input image into its separate channel slices using an ImageJ script.

2) Deconvolve the individual channel slices using DeconvolutionLab2.

3) Merge the deconvolved slices together into their respective time points.

4) Run the individual time points through CellProfiler.

5) Use the same Python script as the “Create Graph” block to create the graph.

Using a script and producing the intermediate files by hand results in getting some of the results and operations of the original workflow “for free”. As the individual time points are stored as files, the “Select Timepoint” operation corresponds to a simple copy operation. The same happens when storing the cell measurement CSV-file. The image improvement side of the workflow has been dropped from the command line workflow, as the disappearance of the time point selection reduces it to trivial operations, if the used tools have a negligible overhead.

The end result is that the script runs in 1416 s compared to the 1426 s for OPE, which used one compute and one master node on an identical system. The difference of 10 s in runtime is negligible, as the dropped parts of the image improvement workflow side would need at least one more ImageJ script; and ImageJ staring overhead is roughly the same time.

Naturally, the time spent in the external tools is the same for the script and OPE. The differences arise where additional overhead is produced. In the case of OPE, overhead comes from loading libraries, orchestrating the worker nodes, importing inputs, exporting results, and importing/exporting partial data for the use in external tools.

In the case of the script version, the overhead is produced by transforming data into the shape required by the next tool. As ImageJ scripts are used for these transformations, each time data has to be restructured (two times in this case) the cost of starting ImageJ has to be paid.

When comparing both overheads, some of the operations are equivalent: the script has to split the image into individual channels for DeconvolutionLab2. Then, the deconvolved slices have to be re-merged into time points. In comparison, OPE will load the complete image into its internal representation. This internal representation is then exported channel wise for DeconvolutionLab2. The result of this deconvolution is again loaded. As the next operation, this data will the be exported as time points for CellProfiler. The resulting CSV files have to be imported, stitched into one large table, and re-exported for the graph generation. The resulting graph image is then imported and immediately exported as the final result. There also is the additional work in the image improvement side. However, it is small, as the involved operations are implemented as plug-ins and do not require the export of data.

A graphical representation of this overhead analysis is shown in Figure 7.9. Here overhead resulting from transforming the complete image is represented by solid blue and green boxes. Overhead from handling tables or single plane images is visualised using patterns of the same colour. As described above, the overhead for transforming the complete image is the same in both cases, even though it occurs in different steps. The other overhead portions handle data of a much smaller size. It is to be expected that their influence on the runtime is thus considerably less pronounced.

A place where OPE saves time in comparison to the command line script is when loading the required libraries once. These are loaded only once, when an operation

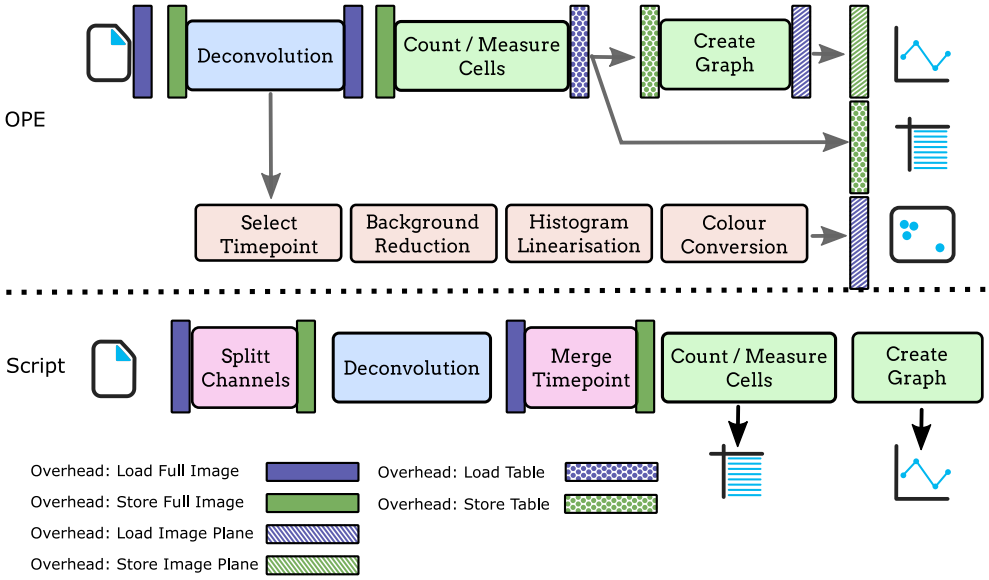


Figure 7.9: Overhead comparison between OPE and the command line script

requires them the first time. In contrast, each time ImageJ is used, it will go through its initialisation, loading libraries such as Bio-Formats. As a result, the non-trivial staring time for ImageJ accumulates over multiple operations.

Finally, the time needed to construct the command line script has to be considered. To create the script presented here, not only the script itself, but also two ImageJ scripts had to be created. If the use of scripts is an established practice, it can be assumed that ImageJ scripts are already present as commonly used helper functions. However, depending on the granularity of each of this helper scripts the performance of the final script can severely be impacted, as each use incurs the start-up cost of the used tools. As assumed in the design of OPE, most tools will work on images. Each time such a tool is used, an image library is loaded on top of parsing and loading the image into memory.

Additionally, the shell script would need to include proper handling of the files and directories used in its steps. As OPE relies on plug-ins and uses OMERO both aspects are provided with minimal overhead for the user.

7.7 Reproducibility

To ensure reproducibility, the results of the workflow are automatically annotated with reproducibility meta-data. Inside OMERO this information is stored as an attachment. These attachments do not clutter the UI as they are somewhat hidden. However, it is easy to access all attachments of an object through the API provided by OMERO.

An example of the reproducibility information produced by the execution of the example workflow is shown in Listing 7.1. It is a JSON formated text file containing the original workflow in the "blocks" and "links" sections, as well as information about this particular execution of the workflow in the "parameters" and "versions" sections.

The "blocks" section contains all used work block. The example block "OmeroImageInputBlock" in lines 6 to 11 shows the data for a block to load an image from OMERO. The corresponding parameter used in this execution is situated in the "parameters" section, lines 23 to 29. As the parameter for this work block was an image, additional information was recorded as discussed in section 4.4. This data most prominently includes the file name, size, and hash. The other shown work block in lines 13 to 16 represents the deconvolution. As this plug-in is version controlled it requires a Git address in lines 14 and 15 as well as the version used in this execution in line 31 of the "versions" section.

```
1  {
2    "name": "ApplicationExample",
3    "runId": "1bf9ba1d-5962-4e93-af5f-5cf19d5d245a"
4    "blocks": [
5      {
6        "elementId": "0"
7        "blockName": "Load Image", // display name
8        "blockId": "OmeroImage",
9        "blockType": "de.c3e.ProcessManager.OmeroImageInputBlock", // class name
10       "Inputs": ["ImageId"], "Outputs": ["LoadedImage"],
11       "positionX": 355, "positionY": 157, // layout information
12       ...  },
13     {"blockName": "Deconvolution", // versioned block
14      "GitFilePath": "ParallelPlaneDeconvolutionTool.java",
15      "GitRepo": "https://git.inf-ra.uni-jena.de/xo46rud/OpePlugins.git",
16      ...  },
17     ... // more blocks
18   ],
19
20   "parameters": [ // supplied parameters
21     ["3", "TimeIndex", "2", "in"], // elementId, input name, value, input/output
22     ...
```

```
23      ["0", "Value", "10TStackNiceCell3ConvedNoiseV2.tif", "out",  // file parameter
24        "FileInfo:
25        File: "10TStackNiceCell3ConvedNoiseV2.tif"
26        Size: 7510270
27        Hash sha256sum: 23ba5ffe78500fdc6ac6af67f6168d57f55a0246d166f2fedd6caf267217fa1e
28        ... // more file information" ,
29        "OMERO ID:1366"],
30    "versions": [ // elementId , commit hash
31      ["1", "a48ede8dbb81f8f8ca3934f228c3665ca1bd8d1a"]
32      ],
33    "links": [   // connections between blocks
34          {"sourceBlock": "0", "sourcePort": "LoadedImage", // link origin
35           "targetBlock": "1", "targetPort": "Input", // link destination
36           "anchors": [[1, 0.5, 1, 0, 0, 0], [0, 0.5, -1, 0, 0, 0]]},  // layout information
37          ...
38          ]
39  }
```

Listing 7.1: Reproducibility information, partially simplified and redacted for readability

A user can use this information to understand how the result file was created. OMERO annotations also remain with the image if it should be moved to another project or data set, ensuring that this information is not lost. As JSON is supported by most web programming languages, it is easy to extend the web interface to transfer the reproducibility data into other forms of attachments or other OMERO extensions. As an example, OMERO key-value pair attachments are accessed by the OMERO search functionality. Transferring data about the used work blocks enables the user to search for results generated by a specific tool.

7.8 Conclusion

Comparing the expected and observed behaviour shows that OPE performs as expected and produces the correct result. It successfully utilises multiple compute nodes, despite the individual tools not being optimised for this use case. The timing behaviour derived from the theoretical properties of the workflow is sufficiently close to the actual results. As such, a theoretical model could be created from a workflow automatically to pick an optimal amount of worker nodes participating in the execution.

In comparison to an alternatively created shell script performing the same task, OPE required a comparable amount of time when using a single worker. Also, it took less effort to create the workflow in OPE, requiring no code to be written.

8 Conclusion

To assess OPEs benefit, the three original goals of usability, reproducibility, and performance as presented in section 4.1 are evaluated. Reproducibility is ensured by using workflow descriptions and storing these descriptions, along with the used parameters, tools, and version information as an attachment to each result inside OMERO. Given all this information, a calculation can be repeated at a later time. The use of a version control system ensures that newer tool versions do not hinder the reproducibility of older workflows. Furthermore, all reproducibility related improvements to OMERO benefit and augment OPE.

Performance wise, OPE enables its users to run workflows in a parallel manner, distributed over multiple compute nodes. Furthermore, it is possible to extend this parallelisation onto external tools, even if those tools do not natively support it. Naturally, these improvements are not comparable to a hand-optimised version of a tool. However, the development focus for these tools is often on the used algorithms. The efficient use of the local hardware capabilities is a consideration only after a tool reaches a certain level of maturity. Distributing work over multiple work nodes is the rare exception.

Finally, usability is a difficult criteria to measure. Nevertheless, it is an important topic to ensure, as the primary audience of OPE are biologists without programming experience. To achieve this goal, graphical programming in combination with familiar web iconography and usage patterns is used. The support of external tools also falls in the area of usability. As these tools are offered in the form of simplified graphical blocks, frequent use cases of many tools can be offered to the user through the same usage pattern, without the need to learn each individual tool.

Throughout the design process, additional goals beyond the original ones have been identified, most prominently load balancing and plug-ins. The executed benchmarks show, that the implemented methods achieve the expected results.

Plug-ins are the last area of interest and have proven to be the most complex topic. Even for simple operations, it is hard to strike the right balance between the capabilities of the system and the complexity to implement the plug-in. This

complexity becomes even more apparent when using external tools, as many require the data to be in custom formats and adhere to specific conventions. OPE compensates most of these implementation difficulties by providing adequate fallbacks and conversions. As it stands, OPEs plug-in capabilities are sufficient to cover the used operations and are flexible enough to be extended for further challenges.

9 Outlook

As with the evaluation during the conclusion, potential future improvements in this chapter are grouped according to the three initial goals. When looking at the performance aspects of OPE a range of options and additional features come to mind.

- Implementing a cost function could be used in conjunction with the load balancer data to decide how many nodes should participate in the calculation.
- In its current form, OPE assumes that no node will fail during the calculation. This assumption could be dropped, and techniques like a redundant master node and checkpointing of intermediates could be implemented. However, this effort should only be undertaken if OPE is to be deployed in a more volatile environment.
- Currently, each piece of data is only held by one node. In an effort to parallelise data transfer, intermediate results could be transferred to multiple nodes which would then share the load of the data distribution. This approach is commonly used when implementing a broadcast operation. However, as the network performance of the Ara Cluster was shown to be no bottleneck, this effort should be reserved in case of deployment on another machine.
- Another current assumption of the worker nodes is that each worker will fully utilise its node. While this is true for some external tools, others may not be optimised for a multi-core environment. A solution to this problem could be to start multiple workers on each node and fill these with additional work if possible. Two main possibilities come to mind: either annotate each tool with additional information on how many threads or cores it will use and have the load balancer figure out the resource usage; or have each node spawn multiple workers that are inactive and activate/deactivate these workers according to the current system workload and memory usage.

Usability improvements can take many forms and are somewhat dependent on current web design paradigms, as these define what a user is used to see and is thus familiar with. Some features that would directly improve the usability are as follows.

- A not uncommon, but non-trivial feature would be the possibility to execute and test a workflow with some test data while it is being constructed. Desktop tools like Icy already offer this feature. However, as OPE consists of two separate systems the implementation of a comparable feature is not as straightforward. Especially OPEs capabilities to handle incomplete or incorrect workflows would need to be improved.

- Common web features like search or user management are also currently not present. Particularly user management could benefit from OPEs integration with OMERO, offering the possibility to share workflows with different users.

- The integration into OMERO could also be deepened. In its current state, OPEs imports result images as images and all other results as attachments into OMERO. However, with the right metadata, it would be possible to import tables as OMERO.tables. Such a result is better accessible through OMERO and can be used in conjunction with OMERO's search and processing functionality.

Reproducibility is the last goal set at the inception of OPE. Improvements in this area could focus on either improving the integration of OPEs with other reproducibility projects or extending the recorded metadata to accommodate for other use cases.

- A widespread technique in the field of reproducibility is to use ontologies. Such an ontology is used to precisely express the relationship or meaning of things. OPE could be extended in a way, that the generated workflows and execution protocols are mapped to such an ontology. This way each calculation result will be linked to the meaning of its raw data. This link would enable the use of techniques, such as semantic searches, to better understand the results of experiments.

  Research in this direction is already taking place in the same project this work originated from and resulted in the CAESAR extension for OMERO.

- While calculations expressed by using plug-ins are well documented, the usage of external tools can have some issues regarding version control and deployment. This problem is discussed in section 6.4, which also offered some possible solutions. In particular using one of the recently popular containerisation techniques to wrap tools in a flexible, reproducible, and version controllable fashion would be a great addition to OPE if the accompanied technical problems can be solved.

- Currently, the information saved in the workflow and execution description covers the bare minimum needed to achieve reproducibility. With more complex use cases it could become necessary to record additional information. An example could be more details about the execution environment, or timing information.

Overall, OPE offers a solid basis for running calculations in a user-friendly, reproducible, and parallelised fashion. All of the proposed additions come from extensions of the initially assumed use cases and target audience.

A Plug-in Base Type List

This chapter lists all plug-in interfaces available. This list serves both as an overview as well as an reference for users.

A.1 Plug-ins with constant Dimensions and Sizes

- Base Class `ParallelCalculatorBase<T>`
  Constructor:

```
ParallelCalculatorBase(ImageDimension[] dep)
```

- `SinglePointCalculator<T>`
  Constructor:

```
SinglePointCalculator()
```

  Calculation Method:

```
T Calculate(T data)
```

- `LineCalculator<T>`
  Constructor:

```
LineCalculator(ImageDimension lineDimension)
```

  Calculation Method:

```
T[] Calculate(T[] data)}
```

- `PlaneCalculator<T>`
  Constructor:

```
PlaneCalculator(ImageDimension firstDirection,
                ImageDimension secondDirection)}
```

  Calculation Method:

```
T[][] Calculate(T[][] data)
```

- `CubeCalculator<T>`

  Constructor:

```
CubeCalculator(ImageDimension firstDirection,
               ImageDimension secondDirection,
               ImageDimension thirdDirection)
```

  Calculation Method:

```
T[][][] Calculate(T[][][] data)}
```

- `HyperCube4DCalculator<T>`

  Constructor:

```
HyperCube4DCalculator(ImageDimension firstDirection,
                      ImageDimension secondDirection,
                      ImageDimension thirdDirection,
                      ImageDimension fourthDirection)
```

  Calculation Method:

```
T[][][][]Calculate(T[][][][] data)
```

- `HyperCube5DCalculator<T>`

  Constructor:

```
HyperCube5DCalculator()
```

  Calculation Method:

```
T[][][][][] Calculate(T[][][][][] data)
```

A.2 Projection Plug-ins

- `PointProjector<T>`

  Constructor:

```
PointProjector(ImageDimension projectionDirection)
```

  Calculation Method:

```
T Aggregate(T data1, T data2)}
```

- `PlaneProjector<T>`

  Constructor:

```
PlaneProjector(ImageDimension firstDirection,
               ImageDimension secondDirection,
               ImageDimension projectionDirection)}
```

  Calculation Method:

```
T[][] Aggregate(T[][] data1, T[][] data2)}
```

- `IndexedPointProjector<T>`

  Constructor:

```
IndexedPointProjector(ImageDimension projectionDirection)
```

  Calculation Method:

```
T Aggregate(T data1, T data2,
            int dimIndexData1, int dimIndexData2)}
```

- `IndexedPlaneProjector<T>`

  Constructor:

```
IndexedPlaneProjector(ImageDimension firstDirection,
                      ImageDimension secondDirection,
                      ImageDimension projectionDirection)
```

  Calculation Method:

```
T[][] Aggregate(T[][] data1, T[][] data2,
                int dimIndexData1, int dimIndexData2)
```

A.3 External Tool Plug-ins

- `ExternalToolBlockBase`

  Constructor:

```
ExternalToolBlockBase()

ExternalToolBlockBase(SourceControlInfo sourceControlInfo)

ExternalToolBlockBase(SourceControlInfo sourceControlInfo,
                      ToolInputOutputConfiguration ioConfig)
```

  Calculation Method:

```
void RunCalculation(File pathToInputFile, File pathToOutput)
```

- `ParallelExternalToolBase`

  Constructor:

```
ParallelExternalToolBase (ImageDimension[] dep)
ParallelExternalToolBase (SourceControlInfo sourceControlInfo,
                          ImageDimension[] dep)

ParallelExternalToolBase (SourceControlInfo sourceControlInfo,
                          ToolInputOutputConfiguration ioConfig,
                          ImageDimension[] dep)

ParallelExternalToolBase (SourceControlInfo sourceControlInfo,
                          ToolInputOutputConfiguration ioConfig,
                          IMergePostProcessing modifier,
                          ImageDimension[] dep
```

  Calculation Method:

```
void RunCalculation(File pathToInputFile, File pathToOutput)
```

Acronyms

API Application Programming Interface

CLI Command Line Interface

CSV Comma-Separated Value

CAESAR CollAborative Environment for Scientific Analysis with Reproducibility

DMA Direct Memory Access

DOI Digital Object Indentifier

FCS Fluorescence Correlation Spectroscopy

DFG German Research Foundation

GUI Graphical User Interface

GPU Graphics Processing Unit

HPC High Performance Computing

IDE Integrated Development Environment

ID Identifyer

IDR Image Data Resource

JIT Just in Time

MPI Message Passing Interface

MIP Maximum Intensity Projection

NIO New Input Output

OOP Object-Oriented Programming

ROI Region Of Interest

RPC Remote Procedure Call

REST Representational State Transfer

PALM Photo-activated Localisation Microscopy

PSF Point Spread Function

SIM Structured Illumination Microscopy

SPIM Selective Plane Illumination Microscopy

SLURM Simple Linux Utility for Resource Management

TERS Tip-Enhanced Raman Spectroscopy

UI User Interface

OPE OMERO Processing Extension

OME Open Microscopy Environment

YARN Yet Another Resource Negotiator

Bibliography

[1] F. Prinz, T. Schlange, and K. Asadullah, "Believe it or not: how much can we rely on published data on potential drug targets?" *Nature Reviews Drug Discovery*, vol. 10, no. 9, pp. 712–712, aug 2011. [Online]. Available: https://doi.org/10.1038/nrd3439-c1

[2] M. Baker, "1,500 scientists lift the lid on reproducibility," *Nature*, vol. 533, no. 7604, pp. 452–454, may 2016. [Online]. Available: https://doi.org/10.1038/533452a

[3] R. D. Peng, "Reproducible research in computational science," *Science*, vol. 334, no. 6060, pp. 1226–1227, dec 2011. [Online]. Available: https://doi.org/10.1126/science.1213847

[4] J. T. Leek and R. D. Peng, "Opinion: Reproducible research can still be wrong: Adopting a prevention approach: Fig. 1." *Proceedings of the National Academy of Sciences*, vol. 112, no. 6, pp. 1645–1646, feb 2015. [Online]. Available: https://doi.org/10.1073/pnas.1421412111

[5] M. R. Munafò, B. A. Nosek, D. V. Bishop, K. S. Button, C. D. Chambers, N. P. du Sert, U. Simonsohn, E.-J. Wagenmakers, J. J. Ware, and J. P. Ioannidis, "A manifesto for reproducible science," *Nature Human Behaviour*, vol. 1, no. 1, p. 0021, jan 2017. [Online]. Available: https://doi.org/10.1038/s41562-016-0021

[6] D. Deutsche Forschungsgemeinschaft, *Vorschläge zur Sicherung guter Wissenschaftlicher Praxis: Empfehlungen der Kommission Selbstkontrolle in der Wissenschaft*. Wiley-VCH, 2006. [Online]. Available: http://www.dfg.de/download/pdf/dfg_im_profil/reden_stellungnahmen/download/empfehlung_wiss_praxis_1310.pdf

[7] A. Casadevall and F. C. Fang, "Reproducible science," *Infection and*

Immunity, vol. 78, no. 12, pp. 4972–4975, sep 2010. [Online]. Available: https://doi.org/10.1128/iai.00908-10

[8] V. Stodden, M. McNutt, D. H. Bailey, E. Deelman, Y. Gil, B. Hanson, M. A. Heroux, J. P. A. Ioannidis, and M. Taufer, "Enhancing reproducibility for computational methods," *Science*, vol. 354, no. 6317, pp. 1240–1241, dec 2016. [Online]. Available: https://doi.org/10.1126/science.aah6168

[9] S. Kanza, C. Willoughby, N. Gibbins, R. Whitby, J. G. Frey, J. Erjavec, K. Zupančič, M. Hren, and K. Kovač, "Electronic lab notebooks: can they replace paper?" *Journal of Cheminformatics*, vol. 9, no. 1, p. 31, may 2017. [Online]. Available: https://doi.org/10.1186/s13321-017-0221-3

[10] J. Potthoff, S. Rieger, P. C. Johannes, and M. Madiesh, "Elektronisches Laborbuch: Beweiswerterhaltung und Langzeitarchivierung in der Forschung," *Digitale Wissenschaft*, p. 149, 2011. [Online]. Available: http://fiz1.fh-potsdam.de/volltext/DigiWis/13471.pdf

[11] M. Rubacha, A. K. Rattan, and S. C. Hosselet, "A review of electronic laboratory notebooks available in the market today," *Journal of Laboratory Automation*, vol. 16, no. 1, pp. 90–98, feb 2011. [Online]. Available: https://doi.org/10.1016/j.jala.2009.01.002

[12] A. Binstock and P. Hill, "Top 9 best electronic lab notebooks review," april 2012. [Online]. Available: http://splice-bio.com/the-7-best-electronic-lab-notebooks-eln-for-your-research/1

[13] O. Spjuth, E. Bongcam-Rudloff, G. C. Hernández, L. Forer, M. Giovacchini, R. V. Guimera, A. Kallio, E. Korpelainen, M. M. Kańduła, M. Krachunov, D. P. Kreil, O. Kulev, P. P. Łabaj, S. Lampa, L. Pireddu, S. Schönherr, A. Siretskiy, and D. Vassilev, "Experiences with workflows for automating data-intensive bioinformatics," *Biology Direct*, vol. 10, no. 1, aug 2015. [Online]. Available: https://doi.org/10.1186/s13062-015-0071-8

[14] C. Allan, J.-M. Burel, J. Moore, C. Blackburn, M. Linkert, S. Loynton, D. MacDonald, W. J. Moore, C. Neves, A. Patterson *et al.*, "OMERO: flexible, model-driven data management for experimental biology," *Nature Methods*, vol. 9, no. 3, pp. 245–253, feb 2012. [Online]. Available: https://doi.org/10.1038/nmeth.1896

[15] E. Williams, J. Moore, S. W. Li, G. Rustici, A. Tarkowska, A. Chessel, S. Leo, B. Antal, R. K. Ferguson, U. Sarkans *et al.*, "Image data resource: a bioimage data integration and publication platform," *Nature methods*, vol. 14, no. 8, p. 775, jun 2017. [Online]. Available: https://dx.doi.org/10.1038/nmeth.4326

[16] R. Pepperkok and J. Ellenberg, "High-throughput fluorescence microscopy for systems biology," *Nature Reviews Molecular Cell Biology*, vol. 7, no. 9, pp. 690–696, jul 2006. [Online]. Available: https://doi.org/10.1038/nrm1979

[17] R. Wollman and N. Stuurman, "High throughput microscopy: from raw images to discoveries," *Journal of Cell Science*, vol. 120, no. 21, pp. 3715–3722, oct 2007. [Online]. Available: https://doi.org/10.1242/jcs.013623

[18] G. Pau, X. Zhang, A. Kumar, A.-C. Gavin, M. Boutros, and W. Huber, "Automated analysis of high-throughput imaging assays with image HTS," 2011, no longer available, Retrieved 2017 from http://www.ebi.ac.uk/huber-srv/cellmorph/kimorph/.

[19] P. Kankaanpää, L. Paavolainen, S. Tiitta, M. Karjalainen, J. Päivärinne, J. Nieminen, V. Marjomäki, J. Heino, and D. J. White, "BioImageXD: an open, general-purpose and high-throughput image-processing platform," *Nature Methods*, vol. 9, no. 7, pp. 683–689, jun 2012. [Online]. Available: https://doi.org/10.1038/nmeth.2047

[20] A. O'Driscoll, J. Daugelaite, and R. D. Sleator, "'Big data', Hadoop and cloud computing in genomics," *Journal of biomedical informatics*, vol. 46, no. 5, pp. 774–781, oct 2013. [Online]. Available: https://doi.org/10.1016/j.jbi.2013.07.001

[21] A. Welivita, I. Perera, and D. Meedeniya, "An interactive workflow generator to support bioinformatics analysis through GPU acceleration," in *2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, nov 2017, pp. 457–462. [Online]. Available: https://doi.org/10.1109/BIBM.2017.8217691

[22] C. Soviany, "Embedding data and task parallelism in image processing applications," Ph.D. dissertation, Delft University of Technology, 2003. [Online]. Available: https://www.semanticscholar.org/paper/Embedding-data-and-task-parallelism-in-image-Soviany/a650ff2f4ec677c9492cdbff0c2c8d25e2237c94

[23] C. Nicolescu and P. Jonker, "A data and task parallel image processing environment," *Parallel Computing*, vol. 28, no. 7-8, pp. 945–965, aug 2002. [Online]. Available: https://doi.org/10.1016/s0167-8191(02)00105-9

[24] G. Cuccuru, S. Leo, L. Lianas, M. Muggiri, A. Pinna, L. Pireddu, P. Uva, A. Angius, G. Fotia, and G. Zanetti, "An automated infrastructure to support high-throughput bioinformatics," in *2014 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, jul 2014, pp. 600–607. [Online]. Available: https://doi.org/10.1109/hpcsim.2014.6903742

[25] J. Leipzig, "A review of bioinformatic pipeline frameworks," *Briefings in Bioinformatics*, p. bbw020, mar 2016. [Online]. Available: https://doi.org/10.1093/bib/bbw020

[26] T. Jenkins, "On the difficulty of learning to program," in *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, vol. 4, sep 2002, pp. 53–58. [Online]. Available: http://www.psy.gla.ac.uk/~steve/localed/jenkins.html

[27] B. A. Myers, "Taxonomies of visual programming and program visualization," *Journal of Visual Languages & Computing*, vol. 1, no. 1, pp. 97–123, mar 1990. [Online]. Available: https://doi.org/10.1016/s1045-926x(05)80036-9

[28] V. Baecker and P. Travo, "Cell image analyzer - a visual scripting interface for ImageJ and its usage at the microscopy facility Montpellier RIO Imaging," in *Proceedings of the ImageJ User and Developer Conference*, vol. 1, jan 2006, pp. 105–110. [Online]. Available: http://dev.mri.cnrs.fr/attachments/40/cell_image_analyzer.pdf

[29] S. Leo, "Enabling data-intensive biomedical studies," Ph.D. dissertation, Universita'degli Studi di Cagliari, 2015. [Online]. Available: http://veprints.unica.it/1196/

[30] F. Taubert and H. M. Bücker, "On the reproducibility of biological image workflows by annotating computational results automatically," in *2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, Nov 2017, pp. 1538–1545. [Online]. Available: https://doi.org/10.1109/BIBM.2017.8217889

[31] ——, "Combining automatic annotation for reproducibility and automatic parallelisation of computations in microscopy image processing workflows," may 2019.

[32] K. W. Eliceiri, M. R. Berthold, I. G. Goldberg, L. Ibáñez, B. S. Manjunath, M. E. Martone, R. F. Murphy, H. Peng, A. L. Plant, B. Roysam, N. Stuurman, J. R. Swedlow, P. Tomancak, and A. E. Carpenter, "Biological imaging software tools," *Nature Methods*, vol. 9, no. 7, pp. 697–710, jun 2012. [Online]. Available: https://doi.org/10.1038/nmeth.2084

[33] T. A. Nketia, H. Sailem, G. Rohde, R. Machiraju, and J. Rittscher, "Analysis of live cell images: Methods, tools and opportunities," *Methods*, vol. 115, pp. 65–79, feb 2017. [Online]. Available: https://doi.org/10.1016/j.ymeth.2017.02.007

[34] A. Chessel, "An overview of data science uses in bioimage informatics," *Methods*, vol. 115, pp. 110–118, feb 2017. [Online]. Available: https://doi.org/10.1016/j.ymeth.2016.12.014

[35] P. P. M. A. Antony, C. Trefois, A. Stojanovic, A. S. Baumuratov, and K. Kozak, "Light microscopy applications in systems biology: opportunities and challenges," *Cell Communication and Signaling*, vol. 11, no. 1, p. 24, mar 2013. [Online]. Available: https://doi.org/10.1186/1478-811X-11-24

[36] J. Schindelin, I. Arganda-Carreras, E. Frise, V. Kaynig, M. Longair, T. Pietzsch, S. Preibisch, C. Rueden, S. Saalfeld, B. Schmid, J.-Y. Tinevez, D. J. White, V. Hartenstein, K. Eliceiri, P. Tomancak, and A. Cardona, "Fiji: an open-source platform for biological-image analysis," *Nature Methods*, vol. 9, no. 7, pp. 676–682, jun 2012. [Online]. Available: https://doi.org/10.1038/nmeth.2019

[37] M. Aron, R. Browning, D. Carugo, E. Sezgin, J. Bernardino de la Serna, C. Eggeling, and E. Stride, "Spectral imaging toolbox: segmentation, hyperstack reconstruction, and batch processing of spectral images for the determination of cell and model membrane lipid order," *BMC Bioinformatics*, vol. 18, no. 1, p. 254, may 2017. [Online]. Available: http://dx.doi.org/10.1186/s12859-017-1656-2

[38] L. Shamir, J. D. Delaney, N. Orlov, D. M. Eckley, and I. G. Goldberg, "Pattern recognition software and techniques for biological image analysis,"

PLoS Computational Biology, vol. 6, no. 11, p. e1000974, nov 2010. [Online]. Available: https://doi.org/10.1371/journal.pcbi.1000974

[39] E. Meijering, O. Dzyubachyk, and I. Smal, "Methods for cell and particle tracking," in *Methods in enzymology*. Elsevier, 2012, vol. 504, pp. 183–200. [Online]. Available: https://doi.org/10.1016/B978-0-12-391857-4.00009-4

[40] Icy. (2019) Icy plugin list. Retrieved 26.02.2019. [Online]. Available: http://icy.bioimageanalysis.org/plugin/list

[41] D. Kundur and D. Hatzinakos, "Blind image deconvolution," *IEEE Signal Processing Magazine*, vol. 13, no. 3, pp. 43–64, may 1996. [Online]. Available: https://doi.org/10.1109/79.489268

[42] L. Cao, P. Juan, and Y. Zhang, "Real-time deconvolution with GPU and Spark for Big Imaging Data Analysis," in *Algorithms and Architectures for Parallel Processing*. Springer International Publishing, dec 2015, pp. 240–250. [Online]. Available: https://doi.org/10.1007/978-3-319-27137-8_19

[43] J. S. Weszka, R. N. Nagel, and A. Rosenfeld, "A threshold selection technique," *IEEE Transactions on Computers*, vol. 100, no. 12, pp. 1322–1326, dec 1974. [Online]. Available: https://doi.org/10.1109/T-C.1974.223858

[44] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE Transactions on systems, man, and cybernetics*, vol. 9, no. 1, pp. 62–66, jan 1979. [Online]. Available: https://doi.org/10.1109/TSMC.1979.4310076

[45] A. Grama, V. Kumar, A. Gupta, and G. Karypis, *Introduction to parallel computing*. Pearson Education, 2003.

[46] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. C-34, no. 10, pp. 892–901, oct 1985. [Online]. Available: https://doi.org/10.1109/tc.1985.6312192

[47] G. L. Taboada, S. Ramos, R. R. Expósito, J. Touriño, and R. Doallo, "Java in the high performance computing arena: Research, practice and experience," *Science of Computer Programming*, vol. 78, no. 5, pp. 425–444, may 2013. [Online]. Available: https://doi.org/10.1016/j.scico.2011.06.002

[48] L. Dalcín, R. Paz, M. Storti, and J. D'Elía, "MPI for Python: Performance improvements and MPI-2 extensions," *Journal of Parallel and Distributed Computing*, vol. 68, no. 5, pp. 655–662, may 2008. [Online]. Available: https://doi.org/10.1016/j.jpdc.2007.09.005

[49] Oracle, "Bytebuffer (java platform se 7)," 2006. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html

[50] T. S. community, "Numpy internals," jan 2019. [Online]. Available: https://docs.scipy.org/doc/numpy/reference/internals.html

[51] S. Samuel, F. Taubert, D. Walther, B. König-Ries, and H. M. Bücker, "Towards reproducibility of microscopy experiments," *D-Lib Magazine*, vol. 23, no. 1/2, jan 2017. [Online]. Available: https://doi.org/10.1045/january2017-samuel

[52] A. Binstock and P. Hill, "The comparative productivity of programming languages," aug 2012. [Online]. Available: http://www.drdobbs.com/jvm/the-comparative-productivity-of-programm/240005881

[53] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, oct 2000. [Online]. Available: https://doi.org/10.1109/2.876288

[54] D. P. Delorey, C. D. Knutson, and S. Chun, "Do programming languages affect productivity? A case study using data from open source projects," in *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*. IEEE, may 2007. [Online]. Available: https://doi.org/10.1109/floss.2007.5

[55] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. ACM Press, nov 2014, pp. 155–165. [Online]. Available: https://doi.org/10.1145/2635868.2635922

[56] P. Lynch and S. Horton, "Page structure and site design," 2017. [Online]. Available: webstyleguide.com/wsg3/6-page-structure/3-site-design.html

[57] T. G. Mattson, B. Sanders, and B. Massingill, *Patterns for parallel programming*. Pearson Education, 2004.

[58] D. Anderson, "BOINC: A system for public-resource computing and storage," in *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*. IEEE, nov 2004, pp. 4–10. [Online]. Available: https://doi.org/10.1109/grid.2004.14

[59] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1996, vol. 1.

[60] S. Siu and A. Singh, "Design patterns for parallel computing using a network of processors," in *Proceedings. The Sixth IEEE International Symposium on High Performance Distributed Computing (Cat. No.97TB100183)*, IEEE. IEEE Comput. Soc, aug 1997, pp. 293–304. [Online]. Available: https://doi.org/10.1109/hpdc.1997.626434

[61] D. Namiot and M. Sneps-Sneppe, "On micro-services architecture," *International Journal of Open Information Technologies*, vol. 2, no. 9, pp. 24–27, 2014. [Online]. Available: http://injoit.org/index.php/j1/article/view/139

[62] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables DevOps: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, may 2016. [Online]. Available: https://doi.org/10.1109/ms.2016.64

[63] B. Randell, P. Lee, and P. C. Treleaven, "Reliability issues in computing system design," *ACM Computing Surveys*, vol. 10, no. 2, pp. 123–165, jun 1978. [Online]. Available: https://doi.org/10.1145/356725.356729

[64] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 23–31, jan 1987. [Online]. Available: https://doi.org/10.1109/tse.1987.232562

[65] J. Dunkel and A. Holitschke, *Softwarearchitektur für die Praxis*. Springer-Verlag, 2013.

[66] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[67] S. Samuel and B. König-Ries, "Reproduce-me: Ontology-based data access for reproducibility of microscopy experiments," in *European Semantic*

Web Conference. Springer, nov 2017, pp. 17–20. [Online]. Available: https://doi.org/10.1007/978-3-319-70407-4_4

[68] S. Samuel, "Integrative data management for reproducibility of microscopy experiments," in *European Semantic Web Conference*. Springer, may 2017, pp. 246–255. [Online]. Available: https://doi.org/10.1007/978-3-319-58451-5_19

[69] S. Samuel, K. Groeneveld, F. Taubert, D. Walther, T. Kache, T. Langenstück, B. König-Ries, H. M. Bücker, and C. Biskup, "The story of an experiment: a provenance-based semantic approach towards research reproducibility," in *Proceedings of the 11th International Conference Semantic Web Applications and Tools for Life Sciences, SWAT4LS 2018*, dec 2018. [Online]. Available: https://doi.org/10.6084/m9.figshare.7345865.v2

[70] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on software engineering*, vol. 14, no. 2, pp. 141–154, feb 1988. [Online]. Available: https://doi.org/10.1109/32.4634

[71] A. Darte, Y. Robert, and F. Vivien, *Scheduling and automatic Parallelization*. Springer Science & Business Media, 2012.

[72] S. Sharma, S. Singh, and M. Sharma, "Performance analysis of load balancing algorithms," *World Academy of Science, Engineering and Technology*, vol. 38, no. 3, pp. 269–272, 2008. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.307.1711

[73] B. Meyer, *Object-oriented software construction*. Prentice Hall New York, 1988, vol. 2.

[74] M. Fowler. (2004) Inversion of control containers and the dependency injection pattern. Retrieved 26.02.2019. [Online]. Available: https://martinfowler.com/articles/injection.html

[75] S. Meyers, *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*, 1st ed. O'Reilly Media, Inc., 2014.

[76] Oracle, "Type erasure (the java tutorials > learning java language > generics (updated))." [Online]. Available: http://docs.oracle.com/javase/tutorial/java/generics/erasure.html

[77] O. Nautsch, "Class von type parameter," nov 2008. [Online]. Available: http://www.nautsch.net/2008/10/28/class-von-type-parameter-java-generics/

[78] StackOverflow, "How to get a class instance of generics type t," aug 2010. [Online]. Available: https://stackoverflow.com/questions/3437897/how-to-get-a-class-instance-of-generics-type-t

[79] G. Hutton, "A tutorial on the universality and expressiveness of fold," *Journal of Functional Programming*, vol. 9, no. 4, pp. 355–372, jul 1999. [Online]. Available: https://doi.org/10.1017/S0956796899003500

[80] R. Chamberlain and J. Schommer, "Using Docker to support reproducible research," *figshare*, jul 2014. [Online]. Available: https://doi.org/10.6084/m9.figshare.1101910

[81] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, jan 2015. [Online]. Available: https://doi.org/10.1145/2723872.2723882

[82] J. Hwee, "Evaluating virtualization and workflow management systems for biomedical application to increase computational scientific reproducibility," *Scholar Archive*, sep 2015. [Online]. Available: https://doi.org/10.6083/m49p30mq

[83] F. Moreews, O. Sallou, H. Ménager, Y. Le bras, C. Monjeaud, C. Blanchet, and O. Collin, "Bioshadock: a community driven bioinformatics shared docker-based tools registry," *F1000Research*, vol. 4, dec 2015. [Online]. Available: https://doi.org/10.12688/f1000research.7536.1

[84] B. Pavie, "Clipart," 2015. [Online]. Available: openclipart.org/user-detail/ben

[85] H. Kirshner, F. Aguet, D. Sage, and M. Unser, "3-D PSF fitting for fluorescence microscopy: implementation and localization application," *Journal of microscopy*, vol. 249, no. 1, pp. 13–25, nov 2012. [Online]. Available: https://doi.org/10.1111/j.1365-2818.2012.03675.x

[86] K. Kvilekval, D. Fedorov, B. Obara, A. Singh, and B. S. Manjunath, "Bisque: a platform for bioimage analysis and management," *Bioinformatics*, vol. 26, no. 4, pp. 544–552, dec 2009. [Online]. Available: https://doi.org/10.1093/bioinformatics/btp699

[87] R. S. Wilson, L. Yang, A. Dun, A. M. Smyth, R. R. Duncan, C. Rickman, and W. Lu, "Automated single particle detection and tracking for large microscopy datasets," *Royal Society Open Science*, vol. 3, no. 5, p. 160225, may 2016. [Online]. Available: https://doi.org/10.1098/rsos.160225

[88] R. C. Martin, "The dependency inversion principle," *C++ Report*, vol. 8, no. 6, pp. 61–66, 1996.

[89] ——, "Design principles and design patterns," *Object Mentor*, vol. 1, no. 34, p. 597, 2000.

[90] O. P. Hamill, A. Marty, E. Neher, B. Sakmann, and F. Sigworth, "Improved patch-clamp techniques for high-resolution current recording from cells and cell-free membrane patches," *Pflügers Archiv*, vol. 391, no. 2, pp. 85–100, aug 1981. [Online]. Available: https://doi.org/10.1007/BF00656997

[91] D. Magde, E. Elson, and W. W. Webb, "Thermodynamic fluctuations in a reacting system—measurement by fluorescence correlation spectroscopy," *Physical Review Letters*, vol. 29, no. 11, p. 705, sep 1972. [Online]. Available: https://doi.org/10.1103/PhysRevLett.29.705

[92] H. Sutter, "Pimples—beauty marks you can depend on," *More C++ gems*, vol. 17, p. 407, may 2000. [Online]. Available: http://www.gotw.ca/publications/mill04.htm

[93] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, K. Thiel, and B. Wiswedel, "KNIME - the Konstanz information miner: version 2.0 and beyond," *AcM SIGKDD explorations Newsletter*, vol. 11, no. 1, pp. 26–31, jun 2009. [Online]. Available: https://doi.org/10.1145/1656274.1656280

[94] D. Blankenberg, G. Von Kuster, N. Coraor, G. Ananda, R. Lazarus, M. Mangan, A. Nekrutenko, and J. Taylor, "Galaxy: a web-based genome analysis tool for experimentalists," *Current protocols in molecular biology*, vol. 89, no. 1, pp. 19–10, jan 2010. [Online]. Available: https://doi.org/10.1002/0471142727.mb1910s89

[95] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat *et al.*, "Taverna: a tool for the composition and enactment of bioinformatics workflows,"

Bioinformatics, vol. 20, no. 17, pp. 3045–3054, nov 2004. [Online]. Available: https://doi.org/10.1093/bioinformatics/bth361

[96] C. A. Schneider, W. S. Rasband, and K. W. Eliceiri, "NIH Image to ImageJ: 25 years of image analysis," *Nature methods*, vol. 9, no. 7, p. 671, jun 2012. [Online]. Available: https://doi.org/10.1038/nmeth.2089

[97] J. Schindelin, C. T. Rueden, M. C. Hiner, and K. W. Eliceiri, "The ImageJ ecosystem: an open platform for biomedical image analysis," *Molecular reproduction and development*, vol. 82, no. 7-8, pp. 518–529, jul 2015. [Online]. Available: https://doi.org/10.1002/mrd.22489

[98] F. De Chaumont, S. Dallongeville, N. Chenouard, N. Hervé, S. Pop, T. Provoost, V. Meas-Yedid, P. Pankajakshan, T. Lecomte, Y. Le Montagner *et al.*, "Icy: an open bioimage informatics platform for extended reproducible research," *Nature methods*, vol. 9, no. 7, p. 690, jun 2012. [Online]. Available: https://doi.org/10.1038/nmeth.2075

[99] E. Afgan, D. Baker, B. Batut, M. Van Den Beek, D. Bouvier, M. Čech, J. Chilton, D. Clements, N. Coraor, B. A. Grüning *et al.*, "The galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update," *Nucleic acids research*, vol. 46, no. W1, pp. W537–W544, may 2018. [Online]. Available: https://doi.org/10.1093/nar/gky379

[100] A. E. Carpenter, T. R. Jones, M. R. Lamprecht, C. Clarke, I. H. Kang, O. Friman, D. A. Guertin, J. H. Chang, R. A. Lindquist, J. Moffat *et al.*, "Cellprofiler: image analysis software for identifying and quantifying cell phenotypes," *Genome biology*, vol. 7, no. 10, p. R100, oct 2006. [Online]. Available: https://doi.org/10.1186/gb-2006-7-10-r100

[101] D. Sage, L. Donati, F. Soulez, D. Fortun, G. Schmit, A. Seitz, R. Guiet, C. Vonesch, and M. Unser, "Deconvolutionlab2: An open-source software for deconvolution microscopy," *Methods*, vol. 115, pp. 28–41, feb 2017. [Online]. Available: https://doi.org/10.1016/j.ymeth.2016.12.015

List of Figures

List of Tables

Listings

Software

Bio-Formats is an open source Java library for reading/writing biological images. It is widely used by microscopy image manipulation tools and maintained in collaboration with microscope manufacturers.
It can be found at openmicroscopy.org.

BisQue (Bio-Image Semantic Query User Environment) [86] is an open source image storage and visualisation tool. It supports many biological image formats and storage solutions such as Amazon Simple Storage Service.
It can be found at bioimage.ucsb.edu/bisque.

BOINC (Berkeley Open Infrastructure for Network Computing) is a framework for distributed computing [58]. It is prominently used by the SETI@home project.
It can be found at boinc.berkeley.edu.

CellProfiler is free open source software-package for detecting and measuring cells and other objects in images [100]. It is written in Java, but depending on the operating system utilises some Jave-Python interop.
It can be found at cellprofiler.org.

DeconvolutionLab2 is a free, open source deconvolution tool for microscopy images [101]. It can be used in combination with ImageJ, but also as a standalone program.
It can be found at bigwww.epfl.ch/deconvolution/deconvolutionlab2/.

Django is a free and open source web framework for Python.
It can be found at djangoproject.com.

Docker is an open source container virtualisation tool, which can be used to isolate certain applications. By providing a very lightweight description, its container can be shared very efficiently. Each such container builds upon other containers and adds the steps needed to install its required software packages.
It can be found at docker.com.

Galaxy is a scientific workflow system commonly used in computational biology [94, 99]. It provides simple processing, as well as complete workflows through a web interface.
It can be found at galaxyproject.org.

Git is a version control system. It can be used to track changes in files and bugs, as well as offer tools used in software development. A range of implementations exists to offer version control for a wide range of potential users, such as GitHub github.com or GitLab gitlab.com.

Hadoop by Apache is a free framework for running distributed processing. It is based on the map-reduce algorithm. It includes its own distributed file system (HDFS) and resource manager (YARN).
It can be found at hadoop.apache.org.

Huygens is a specialised software-package for image deconvolution (see 2.2.1).
It can be found at svi.nl/HuygensSoftware.

Ice byZeroC is a language-independent RPC framework wich can be used to link services writen in different languages. The interfaces of these services are described in a language-independent XML language which Ice uses to generate language-specific templates.
It can be found at zeroc.com/products/ice.

Icy is an open community software for bioimage informatics [98]. While not as popular as ImageJ, it is a powerful and widely used image processing tool.
It can be found at icy.bioimageanalysis.org.

ImageJ is an open source image processing program [96, 97]. It is commonly used by biologists and offers a wide range of functions, as well as plug-in support.
It can be found at imagej.net.
It can also be obtained bundled with many plug-ins in the Fiji [36] package fiji.sc.

IntelliJ is a Java Integrated Development Environment by JetBrains s.r.o. It offers a wide range of features, such as unit testing, library management, and refactoring. It is available in a free community and a commercial version. The development used the free version.
It can be found at jetbrains.com/idea/.

jsPlumb is a JavaScript library to create visual elements and connections on websites. It is available in a commercial and a free community edition. The development used the free version.
It can be found at jsplumbtoolkit.com.

KNIME (Konstanz Information Miner) is a workflow management system [93]. It integrates different data sources with processing modules using graphical programming.
It can be found at knime.com.

LAPACK (Linear Algebra Package) is a software library for numerical linear algebra. It is written in Fortran and can be considered a de-facto standard.
It can be found at netlib.org/lapack/.

LAS (Leica Application Suite) is used to control Leica manufactured microscopes. Additionally, it supports some processing and automation functionality.
It is usually distributed as a part of a microscope system.

matplotlib Is an open source Python library for visualising data.
It can be found at matplotlib.org.

mpi4py Is a Python library implementing the MPI standard [48].
It can be found at bitbucket.org/mpi4py/mpi4py.

OMERO (Open Microscopy Environment) [14] is an open source image storage tool. It supports many biological image formats and offers flexible extension points for further customisation. See also subsection 1.1.2.
It can be found at www.openmicroscopy.org.

Photoshop by Adobe is a commercial image manipulation and raster graphics editor. It is widely used in professional and amateur photography and a de-facto industry standard. It is also used in pre-processing of experimental data, as it offers a wide range of filter and processing tools to improve image quality.
It can be found at adobe.com/photoshop.

SPARK is a programming language for big data processing that can be run distributed.
It can be found at spark.apache.org.

Taverna is a workflow management system [95]. It uses graphical programming to create and visualise workflows consisting of plug-ins, that can be extended by the user.
It can be found at taverna.incubator.apache.org.

ZEN (ZEISS Efficient Navigation), as well as its predecessor AxioVision, is used to control Zeiss manufactured microscopes. Additionally, it supports a limited range of processing and automation functionality. It is usually distributed as a part of a microscope system.
A demo version can be found at zeiss.de/mikroskopie/produkte/.

Glossary

Atomic Force Microscopy (AFM) uses the interaction of attomic forces to generate high-resolution images..

Fluorescence Correlation Spectroscopy (FCS) is a measurement method in which a point of a sample is continuously excited with a laser. When a fluorescent particle enters the area, an increase in the emitted intensity can be detected. By analysing the fluctuation of this intensity over time, properties like diffusion speed can be inferred [91].

Laser Scanning Microscopy (LSM) is a technique to acquire images using a microscope. In it, the sample is scanned by a laser, and pointwise pixel data are created. Because a laser is used, the experimenter can be very precise which fluorescent dye is excited. The created images are confocal, meaning that the information is from a very thin imaging plain, resulting in high-resolution images and the ability to create three-dimensional image stacks. A downside is the relatively slow acquisition rate and the need for a fluorescent molecule, which can require special sample preparation procedures.

Patch Clamp Technique uses a pipette to study the membrane receptors of a cell. Using suction, a piece of the cell membrane is removed and attached to the pipette. This technique commonly measures changes in voltage when exciting the cell.

Photo-activated Localisation Microscopy (PALM), as well as stochastic optical reconstruction microscopy (STORM), is a super-resolution wide-field imaging technique. Both aim to excite only very few fluorescent molecules at a given time point and capture many pictures over a long time frame. Each excited molecule will show as a characteristic spot, the point spread function (PSF), which is specific for the used microscope. If the PSF is known, it can be used

to very precisely locate the molecule, with resolutions higher than those limited by conventional light microscopy. The result is a list of molecule positions, that can be used to create a super-resolution image.

Selective Plane Illumination Microscopy (SPIM) is a light microscopy technique. In it, a sample is illuminated from the side by a thin sheet of light, opposed to from the back in conventional light microscopy. This cross section is then captured by a camera. Because of this, SPIM combines the advantage of creating confocal images with the speed of wide-field microscopy. Since it also needs less light, extended experiment durations with more images become possible. As a typical use case, z-stacks with many slices and from different angles are captured very fast, creating the basis for detailed three-dimensional reconstructions. A disadvantage of this technique is the more involved sample preparation and mounting.

Spectral Imaging is an imaging technique in which captures multiple ranges of the electromagnetic spectrum. In comparison to, for example, traditional cameras which capture information integrated over three colour bands (red, blue, green), a ZEISS LSM 780 can subdivide the light spectrum into 32 consecutive bands of equal size. The advantages of this technique are a more precise representation of the actually colour spectrum. Usually, an image generated by spectral imaging has is processed using a unmixing algorithm.

Structured Illumination Microscopy (SIM) is a light microscopy technique. In it, a sample is illuminated by a specialised structure (stripes), which is moved and rotated. By processing the resulting image, it is possible to generate images with a higher resolution than imposed by the diffraction limit.

The Pointer to Implementation idiom is an implementation technique to achieve inversion of control [88, 89] through a bridge pattern [66]. It can be used to keep the interface of a class constant while making the actual implementation exchangeable. In comparison to an interface, the focus of the idiom is more on hiding implementation details than on providing a set of methods for interaction. See also [92].

Tip-Enhanced Raman Spectroscopy combines AFM methods with Raman Spectroscopy. A small tip is used to scan the surface of a sample using atomic forces to measure the surface. At the same time, a laser is used to gather

additional information about the composition of the sample at the scanning point by exciting molecules and analysing the light that is being emitted as a result.

Unmixing is mathematical technique in which spectral image is processed. The underlying assumption is that an observed spectrum is the (linear) combination of multiple base spectra. As biologist are interested in the distribution of the dye in a given point, unmixing computes the ratio of (known) dye spectra which in there combination result in the observed spectrum.

Wide-Field Microscopy illuminates the sample using a lamp or widened laser. This can be done either by placing it behind or above the sample. The thus created image represents a relatively thick slice. Because of that, it is not suited to generate a three-dimensional reconstruction of the sample. However, it is the easiest microscopy method to use, as it requires little to no specialised sample preparation.